VILNIAUS UNIVERSITETAS
MATEMATIKOS IR INFORMATIKOS FAKULTETAS
PROGRAMŲ SISTEMŲ STUDIJŲ PROGRAMA

Kelių sankryžos prototipo sukūrimas ir skaitinis įvertinimas naudojant “SimPy” diskretinių įvykių simuliavimo biblioteką

Road intersection prototype development and numerical evaluation using the “SimPy” Discrete Event Simulation Library

Baigiamasis magistro darbas

Atliko: Ernestas Zurlys (parašas)

Darbo vadovas: prof. dr. Linas Laibinis (parašas)

Recenzentas: prof. dr. Romas Baronas (parašas)

Vilnius – 2023

TURINYS

LENTELIŲ SĄRAŠAS

PAVEIKSLĖLIŲ SĄRAŠAS

SANTRAUKA

Šio magistrinio darbo tikslas išanalizuoti prototipų naudą Informacinių Technologijų srityje, kokia naudą gali atnešti prototipų darymas IT sistemoms. Tuo pačiu buvo išanalizuota „SimPy“ bibliotekos naudojimas tam tikroms sistemų simuliacijoms kurti ir kokie rezultatai buvo pasiekti naudojant simuliacijos prototipą.

Praktinėje darbo dalyje buvo pasirinkta sukurti dviejų kelių susikirtimo sankryžos simuliacijos aplinkos prototipas. Prototipas buvo sudarytas iš transporto priemonių maršrutų, pėsčiųjų perėjų ir šviesoforo. Šiame prototipe buvo simuliuojamos transporto priemonės ir pėstieji, kad būtų galima surinkti statistinius duomenis, kiek laiko užtrunka jie kiekviename maršrute arba perėjoje. Buvo atlikta statistinė analizė pradinio simuliacijos prototipo ir dviejų šio prototipo pagerinimo variantų. Analizės metu buvo pastebėta, jog sankryžoje pakeitus tik šviesoforo šviesos ciklo laikus buvo sumažintas laukimo laikas lyginant su simuliacijos pradiniu prototipu ir antru pagerinimo variantu. Kai antras pagerinimo variantas buvo simuliuojamas turint po papildomą kelią atlikti kairiuosius posūkius, kas būtų brangiau ir sudėtingiau negu pirmas pagerinimo variantas. Galiausiai buvo atlikti visiems variantams streso testai, kai srautai išauga 5 kartus, šių streso testų pagalba, buvo pastebėta, kad pradinis variantas negalėtų atlaikyti tokio srauto transporto priemonių, išskyrus pirmą ir antrą variantus.

ABSTRACT

The purpose of this master's thesis is to analyze the benefits of prototypes in the field of Information Technology, and what benefits prototyping can bring to IT systems. The SimPy library created system simulations were analyzed and what kind of results they could achieve using simulation prototypes.

In the practical part of the work, it was chosen to create a prototype of the simulation environment of the intersection of two roads. The prototype consisted of vehicle routes, pedestrian crossings, and traffic lights. In this prototype, vehicles and pedestrians were simulated to collect statistics on how long they take each route or crossing. Statistical analysis was performed on the initial simulation prototype and two versions of this prototype improvements. During the analysis, it was observed that after changing only the traffic light cycle times at the intersection, the waiting time was reduced compared to the initial prototype of the simulation and the second improvement option. If the second improvement option was simulated with an additional lane for making left turns, which would be more expensive and more complicated than the first improvement option. Finally, all variants were subjected to stress tests with 5 times increased flow, with the help of these stress tests, it was observed that the original variant could not withstand such a flow of vehicles, except for the first and second variants.

ĮVADAS

Nagrinėjamos temos aktualumas. Dabartinės sistemos yra daug didesnės ir sudėtingesnės negu sistemos, kurios buvo prieš 10 ar 20 metų ir šios sistemos toliau auga, didėja ir tampa sudėtingesnės kurti, išlaikyti ir analizuoti [Com20]. Tai daroma norint pasiekti kuo įmanoma efektyvesnį variantą tam tikrai situacijai, jeigu paimtume gamyklas, galime matyti kaip didžioji dalis rankų darbo yra keičiama į efektyvesnį, kokybiškesnį ir pigesnį būdą kaip automatizavimą būtent to proceso. To tikslas yra sumažinti daromas klaidas, bei lengviau kontroliuoti kokybę kuriamo produkto. Tuo pačiu tai vyksta ir informacijos ir kompiuterių pasaulyje, kuris juda sparčiau į priekį negu kokia kita sritis, būtent todėl esamos sistemos, bei įranga yra keičiama ar tobulinama labai sparčiai, nes norima pasiekti kuo įmanoma daugiau naudojant naujas ar atnaujintas sistemas. Tačiau tai turi savų problemų, kai sistemos tampa labai sudėtingos ir turi daug priklausomybių nuo kitų sistemų, jų palaikymo ir kūrimo išlaidos išauga nežmoniškai ne tik pinigų atžvilgiu, bet ir prietaisų, kompiuterinės įrangos ir kitos dedikuotos įrangos kaina. Keletas šių išlaidų būtų: užtikrinti įrangos komponentų spartumą, atminties kiekiai ir tipai, susiekimo laikas, informacijos kokybės, klaidų pralaidumas ir kitos išlaidos. Todėl dabartinėms kokybiškoms sistemos sukurti neužtenka atlikti nefunkcinių reikalavimų užtikrinimą testais, nes ne visada gausime efektyviausią ir patikimiausią atvejį. Todėl rekomenduojama naudoti papildomus įrankius kurie padeda įsitikinti ir lengvai pakeisti, kad būtų pasiekta efektyvesnė, patikimesnė ar geresnė kuriamos sistemos konfigūracija, bei gautume informaciją, kuri sistemos dalis yra problematiška ir reikėtų į ją kreipti daugiau dėmesio ar priimti naują sprendimą tam išspręsti. Vienas iš tokių įrankių būtų Python kalba parašyta „SimPy“ [TSP20] biblioteka, kurios tikslas yra diskrečių įvykių simuliavimo aplinka, kuri leidžia simuliuoti kuriamos sistemos prototipą ar jau esančios sistemos veikimą, aprašant joje vykstančius įvykius, komunikacijas tarp komponentų, galima aptikti galimus gedimus, modeliuoti laiko pauzes ir daugiau. Naudojant sukurtos sistemos prototipo simuliavimus yra gaunama galimybė statistiškai įvertinti sistemos darbo charakteristikas kaip: efektyvumas, patikimumas, veikimo kokybė ir daugelis kitų, žinant šias charakteristikas galima palyginti skirtingas sistemos konfigūracijas, kad sužinotume kokia yra geresnė ir labiau tinkama esančiam atvejui. Todėl sistemos simuliacijos įrankiai yra labai svarbūs ir naudingi kurti naujoms sistemoms, bei patobulinti esančias sistemas, bet žinoma šie įrankiai gali būti naudojami ne tik programų sistemoms pagerinti ir tobulinti. Pasaulyje yra daugelis kitų procesų ir atvejų, kur simuliacijos įrankiai gali praversti norint pagerinti tam tikrą procesą, veiklą, sumažinti išlaidas, padidinti pelną ir daugelį kitų atvejų. Dažnai būna, kad norint atlikti tam tikrą analizę gali kainuoti

pakankamai daug resursų ir išlaidų, bei kartais pakeisti naują įrangą nėra taip lengva ir prieinama, todėl dažnai žmonės atlieka analizę kaip būtų galima pagerinti tam tikrus procesus. Būtent čia simuliacijos įrankis kaip „SimPy“ labai praverčia, nes proceso simuliaciją galima atlikti pakankamai lengvai ir pigiai. Įrankio pagalba užtenka sukurti tik prototipą, kad būtų galima analizuoti sistemos veikimą. Todėl tai dažnai gali būti lengvesnis ir efektyvesnis būdas atlikti analizę (keletas pavyzdžių: įrankių taisymo simuliacija, detalių gaminimo brokuotų prekių apskaičiavimas, sistemos duomenų siuntimo užtrūkimas ir kt.). Simuliacijos įrankių tikslas yra padėti atrinkti tai kas yra svarbiausia tam tikroje sistemoje, kartu gali pateikti naujų sprendimų, kurių tikslas pagerinti tam tikrą veiklą ar procesą.

Darbo objektas. Šio darbo objektas – kelių sankryžos sistemos prototipo diskrečių įvykių simuliacija ir sistemos statistinis įvertinimas.

Darbo tikslas. Šio darbo tikslas – sukurti kelių sankryžos sistemos prototipą naudojant „SimPy“ biblioteką, atlikti sukurtos sistemos simuliaciją ir statistinį įvertinimą naudojant diskrečių įvykių simuliavimo metodą.

Darbo uždaviniai. Darbo tikslui pasiekti iškelti šie uždaviniai:

1. Atlikus literatūros apžvalgą, identifikuoti, kur dažniausiai naudojami sistemų simuliavimo įrankiai, kokios charakteristikos yra svarbiausios, kokie pagerinimai buvo pasiekti naudojant tokio tipo įrankius.
2. Išanalizuoti „SimPy“ įrankį, sujungti šį simuliacijos įrankį su Python biblioteka „Matplotlib“, panaudoti simuliacijos rezultatus duomenų analizei, duomenų kaupimui ir rezultatų vizualizacijai.
3. Sukurti kelių sankryžos sistemos prototipo simuliaciją naudojant pasirinktą įrankį, detaliai paaiškinti simuliacijos struktūrą ir veikimą, simuliuojamos sistemos parametrus (konfigūracijas), bei statistinių duomenų kaupimą.
4. Atlikti sukurtos sistemos simuliacijos statistinį įvertinimą, išanalizuoti gautus rezultatus, patikrinti galimus naujus simuliacijos atvejus, bei palyginti skirtingas tokių sistemų konfigūracijas.

Darbo vertė. Šio magistro darbo tikslas panaudoti diskrečių įvykių simuliavimo programinį įrankį kelių sankryžos sistemos prototipui sukurti ir jį išanalizuoti. Šio įrankio pagalba galima išsiaiškinti tam tikras sistemos prototipo charakteristikas (efektyvumas, patikimumas, kokybė,

klaidų pralaidumas, greitis ir kt.), todėl šis įrankis gali būti naudojamas daugiau negu tik programų sistemų prototipams, bet ir atvejams, kur galima simuliuoti tam tikrą procesą. Atlikus simuliavimą bus atlikta statistinė analizė, kurios metu bus pateikiama informacija apie šią sistemą ir iš to bus galima matyti kokia nauda gali būti pasiekta naudojant šį įrankį. Tuo pačiu simuliacijos analizės metu surinktų duomenų ir informacijos pagalba bus galima matyti ko galima tikėtis iš šios sistemos, jeigu ji būtų sukurta pagal prototipą. Šios informacijos ir duomenų turėjimas gali padėti greitai sukurti skirtingas sistemos konfigūracijas, kad būtų galima surasti geresnį sistemos variantą negu pradinis.

1. Analitinė dalis

Analitinėje dalyje analizuojami tokie moksliniai šaltiniai kurie yra susiję su ruošiama magistrinio darbo tema. Magistrinio darbo temą galima išskaidyti į tris pagrindines dalis, todėl visi moksliniai šaltiniai yra susieti su tam tikra magistrinio darbo dalimi. Todėl analitinėje dalyje bus išskirtos trys svarbiausios temos apie kurias bus rašoma detaliau kiekviename iš jų esančiuose poskyriuose. Šios trys pagrindinės temos būtų šios:

1. Sistemų prototipų kūrimas ir kodėl sistemų prototipai yra naudojami informacinių sistemų kūrimo procese ir kokią naudą šie prototipai atneša informacinėms sistemoms. Šiame skyriuje bus nagrinėjami moksliniai darbai apie informacinių sistemų prototipus, kokia gaunama naudą iš jų, bei kaip galima išvengti neigiamų padarinių naudojant prototipus.
2. Diskretinių įvykių simuliavimas, kam tai yra atliekama ir daroma, kokią naudą atneša šie diskrečių įvykių simuliavimo modeliai informacinių sistemų kūrimo procese.
3. Diskrečių įvykių simuliavimo įrankiai, kur ir kaip buvo naudojamas „SimPy“ įrankis, bei detaliau apie patį „SimPy“ įrankį kuris bus naudojamas atlikti šiam magistro darbui.

Analitinės dalies skyrių sudaro šios trys temos, kurios bus paremtos moksliniais darbais ir kitais patikimais šaltiniais. Pačiame magistro darbo dokumente šios temos bus pavadintos būtent taip:

1. Sistemų prototipų analizė.
2. Diskrečių įvykių simuliavimo metodų analizė.
3. „SimPy“ įrankio analizė.

1.1. Sistemų prototipų analizė

Pats pirmas dalykas, kurį reikia išsiaiškinti yra kodėl kuriami programų sistemų projektai žlunga ir ar prototipai padėtų išspręsti tam tikrus trūkumus ir klaidas? Šiam klausimui atsakyti buvo naudojamas labai geras mokslinis šaltinis apie informacinių technologijų projektų klaidas ir šių klaidų sprendimus [Lau20]. Šiame šaltinyje buvo atlikti tyrimai, kurių metu buvo išsiaiškinta, kodėl žlunga IT projektai, kokios tai yra priežastys, bei kaip išspręsti šias problemas. Tyrimas buvo atliktas Danijoje su tam tikrais penkiais dideliais IT projektais, kurie yra: elektroninė žemių registro sistema, elektroninė kelionių kortelių sistema, policijos bylų tvarkymo sistema, skolų surinkimo sistema ir sveikatos sąrašų sistema, visos šios sistemos nėra mažos ir visos jos turėjo daugelį problemų. Moksliniame darbe buvo rasta net 37 problemos ir kiekvienai iš jų buvo pateikti sprendimai, kurie buvo aptarti su specialistais. Visos šios problemos buvo suskirstytos į grupes ir kiekvienai problemai buvo pasiūlytas sprendimas, jeigu toks galimas. Šios problemų sritys ir jų kiekis buvo toks:

- Analizė (angl. *Analysis*), šioje grupėje buvo rasta 10 skirtingų problemų, kurios buvo priežastis, kad IT projektas turėjo tam tikrų problemų: laiko atžvilgiu, funkcionalumu, biudžeto atžvilgiu ir panašiai. Šioje grupėje pagrindinės problemos buvo su reikalavimais ir sprendimais.
- Įgijimas (angl. *Acquisition*), šioje grupėje buvo rasta 3 skirtingos problemos. Šią problemų grupę sudarė blogi laiko pasirinkimai, optimizmas ir blogi kriterijai.
- Projektavimas (angl. *Design*), šioje grupėje buvo rasta 5 skirtingos problemos. Ši grupė yra pati svarbiausia šio magistrinio darbo atžvilgiu, nes beveik visos šios problemos, kurios buvo nustatytos turi vieną sprendimą kuris yra: išankstinis prototipų kūrimas su testavimu. Šios problemos bus aprašytos detaliau vėliau šiame po skyriuje.
- Programavimas (angl. *Programming*), šioje grupėje buvo rasta 2 skirtingos problemos. Šias problemas sudarė: nenumatyti integravimo iššūkiai ir brangūs reikalavimai.
- Testavimas (angl. *Test*), šioje grupėje buvo rasta tik 1 problema. Šią grupę sudaro tik viena problema: kad sistema prieš paleidimą nėra pakankamai ištestuojama.

- Diegimas (angl. *Deployment*), šioje grupėje buvo rasta 3 skirtingos problemos. Šią grupę sudaro: apmokymų trūkumas su pačia sistema, ir blogai numatomi sistemos naudojimo atvejai ir jų laiko užtrūkimas.
- Valdymas (angl. *Management*), šioje grupėje buvo rasta 13 skirtingų problemų. Šią grupę sudaro su valdymu susijusios problemos kaip: nėra verslo tikslų, blogas planavimas, projekto augimo nesustabdymas, problemų nenumatymas ir kitos su valdymu susijusios problemos.

Galima matyti iš darbo rezultatų, jog didžioji dalis problemų kyla būtent projekto pradžioje ir projekto valdymo procesuose, tačiau labai didelės problemos egzistuoja būtent projektavimo stadijoje, nes nekreipia pakankamai daug dėmesio toms problemoms arba jos bandomos išspręsti neteisingai. Būtent šiame moksliniame darbe tai yra išryškinta, jog didžioji dalis projekto programuotojų žino apie tam tikras problemas, bet jos nėra išsiryškinamos dėl blogo projekto valdymo ar planavimo. Tuo tarpu jeigu jos randamos, dažniausiai tai būna sprendžiama per vėlai arba blogai. Kartu reikėtų paryškinti projektavimo problemų grupę detaliau, nes būtent ši grupė yra labiausiai susijusi su informacinių technologijų sistemų prototipų kūrimu ir kokią naudą atneša būtent šie prototipai.

Projektavimo problemų grupę šiame tyrime sudarė 5 rastos problemos, tačiau reikia paminėti tai jog šios problemos ir sprendimai buvo rasti būtent šio tyrimo metu ir aplinkoje. Todėl negalime paneigti jos prototipai skirtinguose projektuose negu šiame tyrime negali išspręsti ir kitas problemas. Apačioje bus pateiktos projektavimo problemos ir pasiūlyti sprendimai kaip galima jų išvengti:

1. Problema: Neužtikrina naudojimosi nors žino kaip tai padaryti (angl. *Doesn't ensure usability, even when they know how*), šios problemos pasiūlytas sprendimas yra: prototipų kūrimas su naudojimosi testais anksti projekto eigoje.
2. Problema: Projektuojami vartotojų naudojimosi langai per vėlai (angl. *Designs user screens too late*), šios problemos pasiūlytas sprendimas yra: prototipų kūrimas su naudojimosi testais anksti projekto eigoje.
3. Problema: Priima sprendimą be jokio supratimo apie jį (angl. *Accepts the solution description without understanding it*), šios problemos sprendimas yra: prototipų kūrimas su naudojimosi testais anksti projekto eigoje.

4. Problema: nemato kiek atliko darbo tiekėjas (angl. *Cannot see how far the supplier is*), šios problemos sprendimas: prototipų kūrimas anksti projekto eigoje ir likusio projekto darbo valandų priežiūra.
5. Problema: Mano būdas nesvarstant tiekėjo būdo (angl. *My way without considering the supplier's way*), šios problemos sprendimas yra: SL-07 reikalavimai (autoriaus pasiūlytas būdas rašyti reikalavimus, kur reikalavimai turi išspręsti problemą, o ne ką sistema turi daryti).

Iš visų šių esamų projektavimo problemų, kurios buvo aptiktos autoriaus tyrime, galima pastebėti jog didžioji dalis jų gali būti išspręsta naudojant prototipus anksti projekto kūrimo eigos procese. Svarbiausias pastebėjimas yra tai, jog šie rezultatai t. y. problemos ir sprendimai buvo aptarti su darbuotojais, kurie dirbo prie šių projektų. Didžioji dalis pateikė, kad žinojo apie šias problemas, ir jų sprendimų būdus tačiau tai nebuvo atliekama ir ignoruojama. Vienintelė išimtis buvo penkta problema apie kurią nieko nežinojo ir nežinojo jos sprendimo būdo. Tačiau iš šio tyrimo galima pastebėti jog didžioji dalis projektų nenaudoja sistemų prototipų anksti projekto kūrimo procese ir iš to iškyla daugelis problemų baigtam produktui. Žinoma būna atvejų, kai tiesiog prototipai nėra kuriami ir yra ignoruojami, kai buvo numatyta šiame tyrime [Lau20].

Iš šio mokslinio tyrimo galime matyti kokią naudą gali atnešti prototipų kūrimas ir ką galime gauti ar išspręsti turėdami juos. Pats autorius paryškina, kad didžioji dalis žino kaip išvengti šių problemų ir ką reikia daryti, bet dažniausiai tai yra blogai suprantama kūrėjų arba jie kuriami blogai. Būtent tai neatneša jokios naudos galutiniai sistemai. Todėl kitoje šio skyriaus dalyje bus analizuojami sukurti sistemų prototipai skirtinguose moksliniuose šaltiniuose ir kas buvo pamatyta iš jų.

1.1.1. Medicinos informacinės sistemos prototipo analizė

Pats pirmas prototipas bus šio šaltinio [AK21], šiame dokumente rašoma apie sukurtą prototipą medicinos informaciniai sistemai. Autorių tikslas buvo sukurti medicinos informacinės sistemos prototipą, kuris būtų susijęs su kitomis sistemomis, būtent lengvai valdomas ne tik naudotojams, daktarams, bet ir pacientams. Šiame darbe daug dėmesio yra dedama kaip ši sistema turi atrodyti, kokios turi būti apsaugos, ką turi daryti ir kiti įprasti dalykai, kurie yra svarbūs kuriamai naujai sistemai. Tačiau patys pagrindiniai tikslai kodėl buvo kuriamas šios informacinės sistemos prototipas yra tokie:

- Racionaliai naudoti ir planuoti dalyvaujančių darbuotojų darbo laiką.
- Visiškai atsikratyti popierinės dokumentacijos, kas leistų sumažinti medicininių klaidų skaičių.
- Padidinti medicinos pagalbos aptarnavimą dėl greito paciento informacijos prieigos.
- Pašalinti informacijos ir tyrimų krypčių dubliavimą.
- Planuoti pacientų apkrovą ir pasiskirstymą medicinos įrangai.

Galima pastebėti, kad šie tikslai yra pakankamai dideli ir platūs, ypač kai kuriama sistema turi būti susijusi su daugeliu kitų esančių organizacijos ar valstybės sistemų. Būtent dėl šios kuriamos sistemos svarbumo ir didumo, norint patikrinti ar kuriama sistema tikrai padės pasiekti šiuos tikslus nėra labai lengva ypač jeigu sistema kuri bus kuriama turi biudžeto ir laiko ribojimus. Todėl tokio pobūdžio sistemai sukurti jos veikimo prototipą ir testuoti tam tikroje aplinkoje būtų lengviausias būdas įsitikinti ar sistema atitinka viską kas buvo iš jos norima ir ar ji padeda pasiekti šiuos tikslus. Šiame darbe buvo sukurta aplikacija, kuri atitiko iškeltus jai reikalavimus ir turėjo didžiąją dalį funkcionalumo, kurių norėjo pasiekti su šiuo prototipu. Šio prototipo tikslas buvo sukurti sistemos pagrindą, kuris galiausiai būtų praplėstas į galutinį produktą. Užtat kaip šis prototipas buvo sukurtas, svarbiausia dalis kaip veikia pati sistema su serveriu gali būti nekeičiama, nes viskas buvo suprojektuota gerai šio prototipo metu. Tai leidžia keisti ir plėsti prototipo funkcionalumą lengviau, negu kuriant visą sistemą iškarto, kas gali išryškinti tam tikras sistemos klaidas, kurios buvo aprašytos šio magistrinio darbo praeitoje dalyje apie prototipų kūrimo svarbą.

Galima iš šio mokslinio darbo analizės matyti, kad sukurtas prototipas gali būti lengvas ir greitas būdas patikrinti kaip turi veikti sistema ir ar ji atlieka tai ko norima iš jos. Būtent prototipo metu galima pamatyti projektavimo klaidas, kurios kartais atsiranda dėl nenumatytų priežasčių. Viena iš tokių nenumatytų priežasčių buvo tokia, kad kuriamos informacinės sistemos vartotojų langų išdėstymas ir veikimas buvo suprojektuoti labai kvalifikuoto žmogaus ir niekas papildomai netikrino ar tai tikrai yra geriausias būdas atvaizduoti sistemą. Tai buvo labai didelė klaida, nes tai įtakojo ir sistemos veikimo funkcionalumą, kai ši problema kilo iš specialisto sukurtos sistemos. Taip buvo sukurta labai sunkiai veikianti sistema, kuri sukeldavo daugiau nemalonumų negu juos ištaisė [Lau20]. Todėl šis kuriamas prototipas medicinos informaciniai sistemai [AK21] buvo svarbiausia kreipti dėmesį į pačius naudotojus ir pagrindinį funkcionalumą, nes jeigu nutiktų tokia

pati problema kaip su minėta specialisto sukurtos sistemos problema [Lau20] ši sistema darytų daugiau blogo negu gero.

1.1.2. Aplinkos apsaugos sprendimų palaikymo sistemos prototipų kūrimo įrankio analizė

Šiame moksliniame šaltinyje buvo kalbama apie aplinkos apsaugos sprendimų palaikymo sistemas ir kaip svarbu turėti atviro kodo (angl. *open-source*) įrankį kurti tokių sistemų prototipams [Yu20]. Pats svarbiausias dalykas yra tai, jog žmonės kurie dirba su aplinkos apsaugos sprendimų palaikymo sistemomis (sutrumpintai AASPS) nėra programuotojai arba turi mažai kompiuterių žinių kurių neužtenka norint kurti AASPS sistemą. Didžioji dalis žmonių šioje srityje yra moksliniai, kurie daugiausiai išmano apie modeliavimą naudojant R kalbą, negu puslapių kūrimo kalbas kurios yra reikalingos sukruti tokio tipo sistemai. Todėl šio darbo autoriai sukūrė ir pasiūlė viešai prieinama įrankį, kurio pagalba kiti aplinkos apsaugos mokslinai galėtų sukurti AASPS sistemos prototipus pigiai ir lengvai. Autoriai išryškina kodėl svarbu turėti tokio tipo įrankį, pagrindinės priežastys yra tokios:

- AASPS sistemos prototipų turėjimas padeda išvengti problemų su suinteresuotomis šalimis, nes neturint jokio sistemos veikimo pavyzdžio, tarkime prototipo, suinteresuotos šalys gali mažiau padėti anksčiau projekto eigoje ir ne visada gali pasiūlyti savo patirties.
- Mažos tyrimo komandos AASPS sistemos projektui neturi pakankamai biudžeto, kad šios sistemos būtų sukurtos užsakovams.
- Mažos tyrimo komandos AASPS sistemos projektui neturi pakankamai techninių žinių šiai sistemai sukurti.

Tai yra pagrindinės priežastys šio įrankio sukūrimui ir pasiūlymui kitiems aplinkos apsaugos tyrėjams. Kartu tokio įrankio turėjimas padėtų sukurti greitai ir kokybiškai AASPS sistemas naudojant greitų prototipų kūrimą. Iš šio mokslinio darbo galima pastebėti kokią naudą gali atnešti kitokio pobūdžio sistemų prototipai, tai pat tai tik pastipriną jog prototipų kūrimas atneša daugiau naudos negu blogo, kartu tai daug pigiau ir geriau negu galutinė sistema, kuri gali neįtikti tiems kas ją naudos.

1.1.3. Prototipų naudojimas daiktų interneto (IoT) srityje analizė

Šis mokslinis darbas yra apie daiktų internetą (angl. *Internet of Things*), šio darbo autoriai yra sukūrę įrankių prototipus, kurių pagalba galima daug lengviau sukurti naujas paskirstytas įterptines sistemas (angl. *distributed embedded systems*) [RVS18]. Daiktų internetas yra labai plati ir vis didėjant rinka, nes kuo toliau tuo daugiau žmonių turi įrenginius (daiktus), kurie yra susiję su internetu. Tokio tipo daiktai kuriami dažniau ir jų patenka į rinką vis daugiau. Tačiau iškyla klausimai: kaip šie daiktai turėtų susisiekti vienas su kitu, kaip juos galima valdyti, ką jie gali atlikti? Iš šių klausimų galima matyti, kad šiems daiktams sistemos gali būti daug platesnės negu numatyta, pavyzdžiui: išmaniosios lemputės (pavyzdžiui: Philips Hue) jos irgi skaitosi kaip daiktų interneto produktas ir jeigu pažiūrėtume, ką su šią lemputę galima daryti, galima pastebėti, kad tai nėra tik paprasta lemputė, bet tai yra lemputė su kuria galima atlikti daug išmanių funkcijų [Phi21] ir visos šios funkcijos turi būti sukurtos tam tikroje sistemoje. Tokio pobūdžio sistemos nėra pigios ir lengvos, todėl šio darbo autoriai sukūrė įrankių prototipą, kurių pagalba galima sukurti paskirstytas įterptines sistemas interneto daiktams valdyti. Pasiūlyti įrankių prototipai apima šias daiktų interneto sritis:

- Techninės įrangos palaikymas.
- Įterptosios įrangos programinė įranga.
- Informacijos saugumo sistemos.
- Interneto technologijas.

Šios keturios skirtingos sritys įeina į daiktų interneto kūrimo procesą ir tam palengvinti buvo sukurti įrankio prototipai, kurių pagalba tai galima atlikti daug greičiau ir pigiau, nes pažiūrėjus kiek skirtingų dalių įeina į daiktų interneto kūrimo procesą galima matyti, kad tai nėra toks lengvas ar pigus projektas. Šie įrankio prototipai buvo naudojami studentų semestro darbams, t. y. studentai turėjo naudoti šiuos įrankius sukurti paskirstytą valdymo sistemą. Po šio bandymo buvo pastebėta, jog studentai sukūrė paskirstytas valdymo sistemas, kurios buvo daug sudėtingesnės negu nenaudojant šiuos įrankius ir jos buvo sukurtos su interneto sąsajomis, kas buvo neįmanoma be šių įrankių. Autoriai pamatę šiuos rezultatus, kad šie prototipai veikė labai gerai ir paspartino sistemų kūrimą, nusprendė sukurti savo įrankius.

Iš šio mokslinio darbo galima matyti, kokią naudą atnešė prototipų naudojamas tikrose situacijose, nes autoriai galėjo išbandyti įrankių prototipus ir gavo teigiamus rezultatus, kad

kuriami įrankiai yra geri. Taip galima teigti, jog prototipų egzistavimas projekto kūrimo procese atneša privalumų norint patikrinti kaip veikia kuriamas produktas ir ar nėra jokių problemų. Bei kartu šiame dokumente buvo aptarta kaip svarbu kurti greitus prototipus paskirstytoms įterptoms sistemoms [RVS18].

1.1.4. Aktyvios pakabos sistemos virtualaus prototipo simuliavimas

Tuo pačiu yra kito tipo prototipai, kurie yra vadinami virtualūs prototipai. Todėl buvo analizuojamas šis mokslinis darbas, kurio metu buvo kuriama mašinos patogi ir kokybiška aktyvios pakabos sistema [MSM08]. Šis darbas yra analizuojamas tam, kad būtų galima pamatyti prototipų naudą ne tik informacinių sistemų kūrimo procese, bet kartu įsitikinti kokią naudą galima gauti iš prototipo simuliavimo proceso, nes šiame magistrinio darbe bus atliekamas kelių sankryžos prototipo kūrimas, simuliavimas, tikrinimas ir optimizavimas. Virtualus prototipas buvo naudojamas ne tik šiame moksliniame darbe [MSM08], bet ir kituose kaip [DRF06] ir [EA16], tik šiuose darbuose virtualaus prototipas buvo naudojamas optimizuoti skirtingiems dalykams naudojant skirtingus metodus. Šio virtualaus prototipo nauda buvo įsitikinta šiame darbe, nes autoriai buvo išsikėlę sau tam tikrus reikalavimus, kuriuos jie norėjo pasiekti su šią aktyvios pakabos sistema, kas suteikia vairuotojui mašinos komfortą. Šiam virtualiam prototipui sukurti buvo naudojama ADAMS programa ir mašinos aktyvios pakabos sistemos simuliacija, kuri irgi vyko šioje pačioje aplikacijoje. Tačiau jie tai bandė tik su savo numatytais parametrais, kurie nėra patys optimaliausi, todėl sistemai pagerinti naudojo neaiškios logikos valdymo simuliaciją. Šiai simuliacijai pagerinti buvo naudojama dar kita aplikaciją MATLAB-ADAMS, kurios metu gavo diagramas. Iš šių diagramų buvo galima matyti virtualaus prototipo simuliaciją ir parametrų optimizavimas padėjo užtikrinti kokybišką ir komfortišką aktyvios pakabos sistemą, kai mašina važiuoja nelygiu keliu.

Iš šio darbo galima pastebėti jog virtualūs prototipai yra naudojami mašinų gamybos rinkoje jau daug metų, nes tai padeda išvengti papildomų išlaidų ir projektavimo pakeitimo išlaidų. Bei kartu buvo rašoma apie simuliaciją ir jos optimizavimą, kad būtų gauti geriausi rezultatai, kas yra labai aktualu šiam magistriniam darbui.

1.1.5. Sistemų prototipų analizės išvados

Apibendrinant galima teigti, kad analizuojamų sistemų prototipų pagalba autoriai galėjo išvengti nepageidaujamų problemų ir kartu buvo pastebėta kur prototipai yra naudojami.

Svarbiausi pastebėjai buvo tie, kad didžioji dalis problemų gali kilti dėl blogo projektavimo, o to išvengti gali padėti ankstyvus sistemos prototipas, tačiau nevisi kreipia tiek daug dėmesio į šiuos sistemos prototipus, nebent tai būna numatyta iš anksto. Tačiau jeigu ir būna numatyti sistemos prototipai jie gali būti atlikti neteisingai ar per vėlai, kai jau reikia pakeisti didžiąją dalį projekto. Taip būna daugeliui projektų ir todėl informacijos technologijų srityje didžioji dalis projektų išeina iš biudžeto ir laiko ribų, bei nutinka taip, kad daug kuriamos sistemos projekto yra keičiama. Šis pats padarinys buvo matomas net ir Amerikos gynybos departamente, kur 47% programų buvo už biudžeto ribų (daugiau negu 25%) ir pradinio biudžeto [DVB19]. Norint sumažinti šias problemas ir norint jų išvengti reikėtų naudotis pasiūlytais sprendimais ir kurti sistemų prototipus anksčiau kūrimo procese [Lau20]. Kartu buvo matoma ir prototipų, kurie buvo sukurti įsitikinti, kad kuriama sistema veikia kaip priklauso, jog nėra jokių didelių problemų [AK21]. Tai pat buvo prototipų įrankių kurių pagalba buvo galima sukurti lengviau ir pigiau AASPS sistemų prototipus [Yu20] ir prototipų įrankių, kurie paspartino darbą studentams [RVS18], kas galiausiai padėjo įsitikinti kuriamų įrankių veiksmingumu. Galiausiai buvo kalbama ir apie virtualius prototipus, kurių pagalba buvo galima įsitikinti kaip turi veikti fizinės sistemos automobiliuose [MSM08] [DRF06] ir robotuose [EA16], bei kaip reikėtų šias sistemas optimizuoti. Todėl galime teigti jog prototipai yra pigus ir lengvas būdas patikrinti tam tikros sistemos veiksmingumą ar bent jos dalį.

1.2. Sistemų Diskretinių Įvykių Simuliavimo analizė

Diskretinių įvykių simuliavimas – tai tam tikra simuliacija, kurios metu sistema yra laikoma kaip diskrečių įvykių kolekcija, kuriose kiekvienas įvykis turi tam tikrą nustatytą efektą visai sistemai. Naudojant tokį principą kiekvienas sistemos procesas gali būti apibrėžtas kaip: proceso įtaką visai sistemai, procese resursų reikalavimais ir proceso įvykiu, kuris gali būti nustatomas iš anksto, nutikti nenumatant arba būti priklausomas nuo kitos sistemos dalies įvykio. Kai visos šios dalys yra apsvarstomos tuomet galima atlikti sukurtos sistemos simuliaciją [BMT21]. Diskretinių įvykių simuliacijos dažniausiai naudojamos tikro laiko sistemose arba tokiose sistemose, kur labai svarbus yra laikas, nes kiekvienas įvykis įvyksta tam tikrame laiko tarpe pačioje sistemoje. Vienas iš tokių pavyzdžių būtų sveikatos priežiūros paslaugų sistema, kuriai diskretinių įvykių simuliavimas padeda išanalizuoti įvykius, kurie gali įvykti su sistema [ASG15]. Pagrindinis tikslas šių simuliacijų yra surasti galimas problemas su sistemomis, nustatyti resursų trūkumus ar nenumatytus įvykius. Diskretinių įvykių simuliavimas padeda priimti loginius sprendimus, ką

reikėtų daryti su sistema, kad ji veiktų geriau, efektyviau arba ką reikėtų pergalvoti, kad sistema būtų geresnė negu yra dabar, nes yra galimybių, kai simuliacija leidžia pamatyti net ir blogiausius atvejus, kas irgi skatintų sprendimų procesą [ASG15]. Simuliacijos rezultatai gali būti tikrinami ir paremiami analitiniais ir kitais būdais, kaip rašoma autorių šiame moksliniame šaltinyje [BGS14], kur autoriai bando išsiaiškinti kaip atsikirti simuliacijos pradinius duomenis nuo išlygintų duomenų. Todėl šie autoriai atlikdami tyrimus su paskirstytomis simuliacijomis, kurios turi vykti daugelį kartų, nustatė jog pradiniai duomenys atsiskiria nuo išlygintų duomenų tik po 126 simuliacijos kartų. O kad būtų gaunami patikimi rezultatai reikėjo atlikti 13506 stebėjimus, tačiau autoriai nori pabrėžti jog šis stebėjimo dydis gali keistis pagal simuliuojamą sistemą. Todėl galima matyti, kad pradiniai simuliacijos duomenys nėra tikslūs ir neturėtų būti vertinami ar analizuojami, nes simuliacijos procese šie duomenys labai skirsis nuo išlygintų duomenų, kurie atspindi simuliacijos procese vykstančius pakeitimus ir atitinka simuliuojamos sistemos veikimo rezultatus, kas būtina norint gauti naudingą informaciją iš šių simuliacijos duomenų. Kas leidžia matyti diskretinių įvykių simuliacijų duodama naudą pačiai sistemai ir sistemų projektuotojų sprendimų procesams, kurie gali būti paremti gauta informacija ir duomenimis iš sistemos simuliacijos naudojant diskretinių įvykių simuliavimo metodus.

Šiame skyriuje bus analizuojama su diskretinių įvykių simuliacija susijusi mokslinė literatūra, kurioje bus analizuojami diskretinių įvykių simuliacijos įrankiai tam tikrose sistemose ir kokią naudą galima gauti iš jų. Tuomet bus analizuojamas atviro kodo diskretinių įvykių simuliacijos įrankiai su komerciniais įrankiais palyginimas. Kartu bus analizuojamas pritaikymo šaltinis iš kurio gausime svarbios informacijos apie atviro kodo diskretinių įvykių simuliavimo įrankių taikymą sistemose, kas parems šio magistrinio darbo atviro kodo diskretinių simuliacijos įrankio „SimPy“ naudojimą norint atlikti sistemos prototipo simuliavimą ir optimizavimą.

1.2.1. Diskretinių Įvykių Simuliacijos darbų parduotuvės sistemoje analizė

Šiame po skyriuje bus analizuojamas sukurtas įrankis „ManPy“, kurio tikslas simuliuoti diskretinių įvykių simuliacijas gamybos procese [OGY14]. Pagrindinis tikslas kodėl buvo kuriamas naujas įrankis vietoj to jog būtų naudojami kiti įrankiai yra tai jog didžioji dalis populiariausių įrankių, kurie padeda simuliuoti sistemas ar procesus naudojant diskretinių įvykių simuliacijas yra labai brangūs ir jų licencijos yra per brangios mažoms kompanijoms, jeigu nori

naudoti šiuos įrankius savo versle. Todėl autoriai nusprendė sukurti savo atviro kodo įrankį, kuris galėtų simuliuoti gamybos proceso simuliacijas, kurių pagalba būtų galima optimizuoti ir valdyti gamybos procesą. Autoriai patikrino 23 atviro kodo diskretinių įvykių simuliacijų projektus, kurie padėtų jiems sukurti savo įrankį savo procesams, vienas iš šių įrankių kuris labiausiai tiko dėl savo bendrumo buvo „JaamSim“. Šis įrankis buvo sukurtas iš pamokų kompiuterio dizaino industrijoje, tačiau yra pakankamai bendro pobūdžio, todėl galima pritaikyti ir kitų procesų simuliavimui. Kitas įrankis buvo „SimPy“, kuris sukurtas naudojant Python kalbą ir yra bendro pobūdžio simuliuoti, bet kokius objektus, kurie yra eilėje. Autoriai pasirinko „SimPy“ diskretinių įvykių simuliavimo įrankį kaip pagrindą savo diskretinių įvykių simuliavimo įrankiui kurti „ManPy“, kuris bus naudojamas gamybos procese. Pagrindinė priežastis šio įrankio pasirinkimui buvo Python kalba, jos efektyvus veikimas ir funkcijų rinkinys simuliuoti sistemos logikai. „ManPy“ įrankis buvo modifikuotas, kad būtų pritaikytas gamybos procese. Šio įrankio veiksmingumui patikrinti, jis buvo naudojamas darbų parduotuvės sistemoje, kad būtų optimizuotas produkto gaminimo laikas, ypač kai darbo parduotuvė dirba su skirtingais užsakymais ir klientais, kitaip sakant gamybos procesas visada keičiasi o tai gali atnešti savų problemų. Autoriai išskyrė pagrindinius privalumus, kuriuos būtų galima pasiekti šioje kompanijoje naudojant diskretinių įvykių simuliavimo įrankio pagalbą:

- Poreikis įvertinti skirtingus gamybos planus prieš priimant sprendimą pagal paskutinius gamybos planus. Tai vienas iš privalumų naudojant diskretinių įvykių simuliavimo įrankius, nes simuliacijų duomenimis galima paremti tam tikrus sprendimus, kaip šiame atvejyje nenumatyta problema gali paveikti visą gamybos procesą.
- Poreikis įvertinti alternatyvius tvarkaraščius, kai gaunami skubūs užsakymai iš klientų ir juos reikia greitai nustatyti šių skubių užsakymų poveikiui nustatyti jau vykdomiems užsakymams. Su diskretinių įvykių simuliavimo įrankio simuliacijos rezultatais galima nustatyti tokius atvejus ir tai leistų priimti apgalvotus sprendimus pasiremiant simuliacijos informacija ir duomenimis.
- Poreikis įvertinti poveikius, kai ištekliai tampa riboti, dėl mašinų gedimo arba projektų vadovų nebūvimo ir kai reikia prognozuoti, kokį poveikį tai turėtų ankstesnių klientų pristatymo įsipareigojimui. Su diskretinių įvykių simuliavimo įrankiu galima nustatyti tokius įvykius: kas gali įvykti simuliacijos metu, taip

gaunama informacija apie šių įvykių poveikį visai sistemai, šiuo atveju gamybos procesui.

Todėl galima sakyti, kad diskretinių įvykių simuliavimo įrankių pagrindinis tikslai šių darbų parduotuvių sistemose būtų: greitai gauti informacija apie esamus užsakymus, kurie turi būti įvykdyti, gauti informaciją apie išteklius kurie yra reikalingi šiems procesams įvykdyti ir simuliuoti gamybos planų pasiskirstymą, kad būtų nustatytas geriausias gamybos planas su mažiausiai delsimų.

Autorių sukurtas diskretinių įvykių simuliavimo įrankis „ManPy“ buvo patikrintas ir išbandytas kompanijos mažoje firmoje, kurioje buvo įsitikinta, jog „ManPy“ įrankis gerai simuliuoja gamybos proceso simuliacijas. Lyginant su kitais diskretinių įvykių simuliavimo įrankiais kaip „SimPy“ ar kitais komerciniais įrankiais, „ManPy“ atliko viską gerai simuliuojant sistemą. Įrankio patikrinimo metu buvo naudojama Skruzdėlių Kolonijos Optimizavimo (SKO) algoritmas, kad būtų nustatytos tinkamiausios taisyklės kaip turi veikti simuliacijoje esantys objektai (mašinos), kad galutinis gaminamo produkto laikas būtų greičiausias. Šiam simuliacijos modeliui buvo sukurtos šios 8 taisyklės, kurias galėjo naudoti simuliacijos metu SKO algoritmas, taip skruzdėlės (simuliacijoje mašinos) galėjo simuliuoti daugelį atvejų su skirtingomis taisyklėmis, kad būtų gaunamas greičiausias gamybos proceso variantas. Šios 8 taisyklės bus pateiktos apačioje:

1. SPT – parenkama užduotis, kuriai reikalingas trumpiausias atlikimo laikas.
2. LPT – parenkama užduotis, kuriai reikalingas ilgiausias atlikimo laikas.
3. MS – parenkama užduotis, kuri turi trumpiausią sąstingio laiką.
4. SRR – parenkama užduotis, kuriai nereikės mašinos kompiuterio paruošimo atsižvelgiant į medžiagų panašumą su praeita užduotimi.
5. ERD – parenkama užduotis su anksčiausia užsakymo data.
6. EDD – parenkama užduotis su anksčiausia termino data.
7. WINQ – parenkama užduotis su trumpiausia eilę iki kito kelionės tikslo.
8. PCO – parenkama kita užduotis pagal vartotojo nustatytą prioritetą.

Tuomet simuliacijos modelyje visoms gamybos procese susijusioms mašinoms buvo paskirta keletas iš šių aštuonių taisyklių. Naudojantis šiomis taisyklėmis simuliacijos metu SKO algoritmas leidžia išrinkti geriausias taisykles gamybos mašinoms pagal gautą darbą. Galiausiai atvaizduoti simuliuojamus gamybos darbus buvo surenkama informacija iš „ManPy“ ir

atvaizduojama kaip darbų tvarkaraštis, kuriame buvo matoma: kokie atliekami darbai, kokios mašinos dirba, kada turi pradėti ar baigti darbą. Tuomet kartu buvo atvaizduojama kita diagrama, kurioje buvo matoma simuliacijos metu naudojamų resursų informacija, kurią sudarė: mašina ir kiek šios mašinos resursų naudojama darbui, laukimui, klaidoms ar neprieinamumui. Tačiau autoriai žino jog ateinantys darbai gali būti skirtingi ir kisti, todėl autoriai sugalvojo būdą, kad darbų užsakymų informacija būtų pateikiama Java Script Object Notraion (JSON) formatu, kuris yra suprantamas „ManPy“ įrankiui ir su šia darbo informacija būtų atliekama jo simuliacija.

Iš šio mokslinio šaltinio [OGY14] galima pamatyti kokią naudą gali atnešti diskretinių įvykių simuliavimo įrankių integravimas gamybos procese. Tuo pačiu galima teigti, jog kitose sistemose ar procesuose šie įrankiai gali atnešti kitų privalumų ypač tam tikrų projektavimo ar pokyčių sprendimuose. Bei papildomi duomenys ir informacija apie sistemos veikimą simuliacijose gali padėti pigiai ir greitai optimizuoti simuliuojamas sistemas naudojant atviro kodo įrankius kaip „SimPy“ ar „ManPy“. Šiame magistro darbe bus naudojamas diskretinių įvykių simuliavimo įrankis „SimPy“ kelių sankryžai simuliuoti ir analizuoti naudojant simuliacijos informacijos žurnalus su duomenų diagramomis.

1.2.2. Diskretinių Įvykių Simuliavimas sveikatos priežiūrai užtikrinti analizė

Šiame po skyriuje bus analizuojamas mokslinis šaltinis apie sveikatos priežiūros kokybės užtikrinimo sprendimų palaikymas naudojant diskretinių įvykių simuliaciją [KBK17]. Šiame darbe autoriai norėjo sukurti sistemą, kuri gali užtikrinti aukštos kokybės sveikatos priežiūrą pacientams, todėl pats svarbiausias dalykas yra užtikrinti aukštos kokybės sveikatos priežiūrą iš valdymo pusės ir atitinkamų asmenų sprendimų. Šis pokytis vyksta nes sveikatos priežiūra pasikeitė iš kiekybinės paslaugos į kokybinę paslaugą, kas reiškia jog reikia paskirstyti turimus išteklius kitaip negu buvo anksčiau. Todėl autoriai šiai sistemai pasiūlyti naudojo tris modeliavimo metodus:

- Diskretinių įvykių simuliavimas.
- Agentų pagrįstas modeliavimas.
- Duomenų analizė.

Visi šie metodai turi savo paskirtis, tačiau šiam magistriniam darbui rūpi tik diskretinių įvykių simuliavimo metodas ir ką autoriai norėjo pasiekti naudojant diskretinių įvykių simuliavimą savo sistemoje. Šie metodai buvo išrinkti tam, kad žmonės kurie yra atsakingų už sprendimų

priėjimą galėtų juos priimti su kiek įmanoma mažiau nesklandumų. Sprendimus pasidarė sunku priimti dėl to, kad sveikatos priežiūros procesai tapo daug kompleksiškesni negu buvo seniau ir kartu buvo sukaupiama daugiau nestruktūrizuotų duomenų. Todėl vienas iš pasiūlytų būdų tai palengvinti buvo naudoti diskretinių įvykių simuliavimą. Šio simuliavimo pagrindinis tikslas darbe buvo simuliuoti medicinos centro veiklą, t. y. įvertinti galimybes ir darbo krūvį kiekvienam departamentui atskirai ir visai ligoninei. Būtent atlikus tokias simuliacijas buvo gauta: departamento darbo krūvių duomenys ir kiekvieno departamento speciali informacija, kurios pagalba galima patikrinti kokybę. Kita priežastis kodėl buvo naudojamos simuliacijos yra tam, kad būtų galima gauti kiekvienos darbuotojų grupės darbo krūvį, kas yra svarbi informacija priimti sprendimams ir optimizacijai darbuotojų tvarkaraščiams. Tuo pačiu tai padėjo išsiaiškinti ligoninės piko valandas, kas labai svarbu ligoninės kokybės užtikrinimui, nes autoriai nustatė jos pagrindinė grupė darbuotojų, kurie užtikrino aukštos kokybės paslaugas buvo 24 valandas dirbančios slaugytojos, kurios juto darbo perkrovas. Šioms simuliacijoms atlikti buvo surinkti duomenys iš HIS (Hospital Information System) ir ACS (Access Control System) apie darbo krūvį šioms slaugytojoms. Atvaizdavus šiuos slaugytojų duomenis diagramoje, kurioje buvo tikrinamas krūvis kiekvieną darbo valandą, buvo nustatyta pagrindinės darbo piko valandos, bei buvo nustatyta jog HIS duomenys nėra tokie patikimi, kaip ACS, nes HIS duomenys turi būti įvesti rankiniu būdu, kas nėra taip tikslu kai ACS automatiškai gauna duomenis, kur slaugytojos buvo paskutinį kartą. Šiems duomenims analizuoti buvo naudojami trys skirtingi metodai:

- Grupavimo metodas – buvo analizuojamas darbo paskirstymas ir buvo rasta jog skirtingos modelio grupės negali priklausyti skirtingoms darbuotojų grupėms, tačiau šis metodas parodė, kad darbo dienų apkrovimas nepriklauso nuo slaugytojos asmenybės ar darbo dienos, bet priklauso nuo darbo krūvio ir specialių atvejų.
- Dažnio analizė – buvo analizuojami skirtingos darbų sekos, kas galėjo parodyti ar yra unikalios darbo pamainos, tačiau buvo rasta jog unikalios darbo pamainos labai retai būna.
- Agentų pagrįstas modeliavimas – buvo analizuojamos skirtingos darbuotojų grupės (aktoriai), šios analizės metu buvo rasta tik darbo piko valandos ir kiek slaugytojos gauna darbo krūvio.

Galima matyti, jog analizuoti ir simuliuoti sunkias sistemas kaip sveikatos priežiūros sistema nėra lengva, tačiau tai padėjo suprasti kada slaugytojos dirba daugiausiai (piko valandos) ir kur

dažniausiai slaugytojos praleidžia savo laiko, kas gali padėti priimti tam tikrus sprendimus, jeigu reikėtų priimti naujus darbuotojus. Visi metodai buvo analizuojami naudojant diagramas ir modelius. Todėl šiame šaltiniame autoriai naudojo diskretinių įvykių simuliacijas išsiaiškinti darbuotojų krūvį, kokie departamentai kiek dirba ir visi šie informacijos duomenys gali būti panaudoti priimti tam tikrus sprendimus pačioje įstaigoje (priimti naujus darbuotojus, paskirstyti darbus ir kiti sprendimai), bei kartu su simuliacijos duomenimis įstaiga gali optimizuoti darbuotojų laiką arba darbo efektyvumą.

1.2.3. Atviro kodo ir komercinio tipo Diskretinių Įvykių Simuliavimo įrankių analizė

Daugelyje mokslinių šaltinių, kuriuose kalbama apie diskretinių įvykių simuliavimo įrankius didžioji dalis darbuotojų ir įmonių kurios šiuos įrankius naudoja būna komercinio tipo [LRM21]. Tai palieką klausimą – koks yra skirtumas tarp komercinių ir atviro kodo diskretinių įvykių simuliavimo įrankių ir kodėl vertėtų naudoti vieną vietoj kito. „ManPy“ įrankio autoriai rekomenduoja naudoti atviro kodo įrankius dėl jų lengvo keitimo ir pritaikymo, tačiau pati pagrindinė priežastis kodėl atviro kodo diskretinių įvykių simuliavimo įrankiai populiarėja ir dažniau yra naudojami negu komerciniai įrankiai yra dėl jų kainos [OGY14]. Komerciniai įrankiai nevisoms kompanijoms ir tyrėjams yra prieinamos dėl jų didelės kainos ir licencijų. Šio šaltinio autoriai pritaria ir labiau gilinasi į atviro kodo ir komercinių diskretinių įvykių simuliavimo įrankių palyginimą gamybos ir logistikos proceso simuliavimui [LRM21]. Autoriai išryškina komercinių įrankių problemą mažoms ir vidutinio dydžio kompanijoms, net ugdymo įstaigoms. Mažoms ir vidutinio dydžio kompanijoms didžiausia problema yra komercinių įrankių kaina, kas gali kainuoti iki dešimčių tūkstančių per metus, tuo tarpu ugdymo įstaigoms yra duodamos dovanos licencijos tačiau jos būna vienaip ar kitaip limituojamos, ką gali daryti su tuo įrankiu. Todėl šio mokslinio šaltinio autoriai nusprendė palyginti 3 populiariausius ir labiausiai atnaujinamus atviro kodo diskretinių įvykių simuliavimo įrankius su 2 komercinio tipo įrankiais, kurie dažnai naudojami gamybos ir logistikos srityse. Atliktuose autorių tyrimuose buvo tikrinami šie penki diskretinių įvykių simuliavimo įrankiai:

1. Arena – pats populiariausias komercinis diskretinių įvykių simuliavimo įrankis, labiausiai naudojamas Šiaurės Amerikoje ir Lotynų Amerikos šalyse. Sukurtas kompanijos The Systems Modelling Corporation and Rockwell Automation.
2. Plant Simulation – kitas populiarus komercinis diskretinių įvykių simuliavimo įrankis, labiausiai naudojamas Vokietijoje kaip standartinis įrankis simuliuoti ir

analizuoti gamybos ir logistikos procesus automobilių industrijoje. Sukurtas kompanijos Siemens.

3. Salabim – atviro kodo diskretinių įvykių simuliavimo įrankis sukurtas naudojant Python kalbą, todėl didžioji dalis funkcionalumo gali būti papildoma naudojant kitas atviro kodo Python bibliotekas ir įrankius. Šis įrankis yra populiariausias duomenų mokslo ir mašininio mokymo srityse.
4. JaamSim – atviro kodo diskretinių įvykių simuliavimo įrankis sukurtas naudojant Java kalbą, todėl didžioji dalis funkcionalumo gali būti papildoma naudojant atviro kodo Java bibliotekas ir įrankius. Šis įrankis labiau naudojamas gamybos ir logistikos srityse dėl geros medžiagos šaltinio pavadinimu „The Big Lean Simulation Library“.
5. CloudSim – atviro kodo diskretinių įvykių simuliavimo įrankis sukurtas naudojant Java kalbą, todėl didžioji dalis funkcionalumo gali būti papildoma naudojant atviro kodo Java bibliotekas ir įrankius. Bei CloudSim turi savo plėtinius kaip CloudAnalyst ir CloudSim Plus. Šis įrankis labiausiai naudojamas kompiuterio mokslo tyrimuose, nes skirtas simuliuoti debesų kompiuterijos programas.

Autoriai atliko visų šių įrankių tyrimą ir gavo kelis pastebėjimus, vienas iš jų toks pats ką buvo pastebėję ir praeito po skyriaus šaltinio autoriai, jog kai kurie komerciniai įrankiai per daug sudėtingi [OGY14]. Kiti pastebėjimai buvo tokie jog visi diskretinių įvykių simuliavimo įrankiai turėjo savų problemų, simuliacijos kūrimo procese, kai kurie įrankiai nebuvo skirti simuliuoti gamybos ir logistikos procesus, kas privedė prie papildomų valandų darbo tai atlikti. Kitas pastebėjimas buvo toks jog buvo planuota naudoti „ManPy“ įrankis, tačiau jis nebuvo atnaujinamas, kas buvo privaloma šių autorių tyrime. Tačiau buvo paryškintas įrankis „SimPy“, kuris yra vienas iš seniausių ir labiausiai žinomų atviro kodo diskretinių įvykių simuliavimo įrankių, bet nebuvo jis žiūrimas dėl modeliavimo neturėjimo, todėl buvo pasirinkta Salabim kitas Python įrankis.

Tyrimo rezultatai yra tokie jog visi atviro kodo diskretinių įvykių simuliavimo įrankiai gali būti naudojami vietoj komercinių įrankių, tačiau reikia atsižvelgti į simuliavimo sritį, nes kiekvienas įrankis turės tam savų privalomų ir trūkumų. Autorių nuomone JaamSim yra geriausias atviro kodo diskretinių įvykių simuliavimo įrankis lyginant su komerciniais šio tipo įrankiais, kai simuliuojami gamybos ir logistikos procesai. Salabim atviro kodo diskretinių įvykių simuliavimo

įrankis autorių atžvilgiu yra labai geras įrankis tiems kurie yra pažengę vartotojai ir moka programuoti, nes naudojant kitas Python bibliotekas šio įrankio funkcionalumas gali būti išplėstas: optimizavimu, vizualizacija, duomenų keitimu, mašinų mokymu ir kitomis bibliotekomis, kas padaro šį įrankį labai lanksčiu lyginant su komerciniais įrankiais. Galiausiai CloudSim atviro kodo diskretinių įvykių simuliavimo įrankis reikalauja labiausiai pažengusio vartotojo iš visų tyrimo įrankių, todėl šis įrankis labiausiai tinka tik informacinių technologijų (IT) specialistams.

Iš šio mokslinio šaltinio tyrimo apie komercinių ir atviro kodo diskretinių įvykių simuliavimo įrankių galima matyti jog atviro kodo diskretinių įvykių simuliavimo įrankiai gali prilygti komerciniams įrankiams funkcionalumu ir nauda. Tai tik parodo jog mažesnės ir vidutinio dydžio kompanijos galėtų įdiegti sistemų simuliacijas savo kompanijose, kad galėtų priimti geresnius sprendimus, kurie yra paremti simuliacijos duomenimis arba galėtų naudoti šiuos įrankius optimizuoti savo sistemą ar dalį jos neišleidžiant didelių pinigų šiai veiklai.

1.2.4. Sistemų Diskretinių Įvykių Simuliavimo analizės išvados

Šiame skyriuje buvo analizuojama daugelis sistemų simuliacijų naudojant diskretinių įvykių simuliavimo įrankius iš kurių buvo nustatyti daugelis privalumų pritaikyti sistemos diskretinių įvykių simuliavimo procesą ir mažoms kompanijoms dėl galimų privalumų. Pagrindinės priežastys kodėl šios simuliacijos buvo atliekamos yra tai jog vystomos sistemos tampa vis didesnės ir sudėtingesnės, kas pasunkina priimti geriausius sprendimus. Todėl didžioji dalis įmonių kurios naudoja diskretinių įvykių simuliavimo įrankius atlikti sistemos simuliacijoms, kad būtų galima išvengti nenumatytų problemų su pačiu sistemos veikimu ar priimti svarbius sprendimus susijusius su sistema pasiremiant simuliacijos duomenimis ir informacija. Tai daroma todėl, nes tai yra lengvas ir pigus būdas įsitikinti tam tikrus sprendimus, kurie gali paveikti sistemą, bei pamatyti nenumatytus problemas ar optimizacijos būdus. Jokia kompanija ypač mažos ir didelės nenorėtų, jog reikėtų keisti sistemą arba paaiškėtų, kad sistema blogai veikia, kai tai buvo nenumatyta, kas privestų prie didelių išteklių sunaudojimo šioms problemoms išspręsti. Tuo pačiu diskretinių įvykių simuliavimo įrankiai buvo naudojami optimizuoti tam tikrus sistemos procesus arba optimizuoti darbuotojų tvarkaraščius, nes visi įvykiai vyksta laiko aibėje, kas leidžia lengvai simuliuoti šiuos procesus ir juos optimizuoti naudojant skirtingus algoritmus. Tai buvo daroma gamybos, logistikos procesų optimizavimui [OGY14] ir medicinos įstaigų darbuotojų tvarkaraščių optimizavimui [KBK17]. Šiuos diskretinių įvykių simuliavimo įrankius naudoja didžiausios įmonės, kad būtų išvengiama didelių išlaidų ir nenumatytų problemų, tačiau nevisos įmonės

naudoja šiuos įrankius, nes jie yra brangūs ir neprieinami mažoms ir vidutinio dydžio kompanijoms, todėl buvo analizuojami atviro kodo diskretinių įvykių simuliavimo įrankiai ar jie gali duoti tokią pat naudą kaip komerciniai įrankiai [LRM21]. Šio tyrimo metu buvo modeliuojama ir simuliuojama gamybos ir logistikos sistema naudojant atviro kodo ir komercinių diskretinių įvykių simuliavimo įrankiai. Buvo pastebėta, kad atviro kodo diskretinių įvykių simuliavimo įrankiai gali taip pat gerai būti naudojami kaip ir komerciniai įrankiai. Šio tyrimo metu autoriai rekomendavo naudoti atviro kodo diskretinių įvykių simuliavimo įrankius dėl jų lengvo keičiamumo, pritaikymo ir kainos. Todėl galima matyti, kad diskretinių įvykių simuliacijos gali būti nebrangiai pritaikytos daugeliui skirtingų sistemų, kurių metu gaunami duomenis, informacija apie simuliuojamas sistemas, kas padėtų priimti svarbius sprendimus, pakeitimus ar optimizuoti tam tikrus procesus.

1.3. „SimPy“ Diskretinių Įvykių Simuliavimo įrankio analizė

„SimPy“ tai yra atviro kodo diskretinių įvykių simuliavimo įrankis sukurtas Python kalba ir išleistas MIT (Massachusetts Institute of Technology) licencija. Python kalba yra labai populiari ir turi daugelį skirtingų bibliotekų, kurios gali būti panaudojamos kartu su „SimPy“ biblioteka funkcionalumui praplėsti, todėl viena iš populiariausių modeliavimo Python bibliotekų kaip „Matplotlib“ gali būti integruota simuliacijos duomenims atvaizduoti, kas palengvintų duomenų analizės procesą. „SimPy“ veikimas susidaro iš 4 dalių, kurios reikalingos atlikti sistemos diskretinių įvykių simuliavimo simuliacijas:

- Simuliavimo aplinkos (angl. *Environments*).
- Simuliavimo procesai (angl. *Process*).
- Simuliavimo įvykiai (angl. *Events*).
- Simuliavimo bendrai naudojami resursai (angl. *Shared Resourses*).

Simuliavimo aplinka valdo simuliacijos laiką, tai pat valdo simuliacijos įvykių planavimą ir apdorojimą savo vidiniame laikrodyje. Simuliacijos gali vykti tol, kol nebeliks simuliacijos įvykių, kol bus pasiektas tam tikras simuliacijos laikas arba kol įvyks tam tikras simuliacijos įvykis. Šis simuliacijos aplinkos lankstumas leidžia vykdyti simuliacijas tiek laiko kiek reikia arba kol bus matomas tam tikras įvykis. Kiekviena simuliacijos aplinkos valdymo funkcija yra pateikta 1 lentelėje.

1 lentelė. Simuliacijos aplinkos pagrindiniai metodai ir jų poveikis simuliacijai

Simuliacijos aplinkos veikimo metodai	**Metodo poveikis simuliacijai**
env.run(), env – simuliavimo aplinka, *run()* – metodas paleidžiantis simuliacijos aplinką.	Paleidžiama ir pradedama simuliuoti visus simuliacijos įvykius, kol bus pabaigta simuliacija. Jeigu *run()* metodas yra tuščias ir simuliacija neturi išėjimo ji gali vykti visą laiką.
env.run(until=x), env – simuliacijos aplinka, *run()* – metodas paleidžiantis simuliacijos aplinką, *until=x* parametras, kai *x* yra simuliacijos trukmės laikas.	Paleidžiama ir pradedama simuliuoti visi simuliacijos įvykiai tol kol simuliacijos vidinis laikrodis pasieks *x* laiką, tuomet simuliacija yra sustabdoma.
env.run(until=procesas), env – simuliacijos aplinka, *run()* – metodas paleidžiantis simuliacijos aplinką, *until=procesas* parametras, kai *procesas* yra simuliacijos įvykis.	Paleidžiama ir pradedama simuliuoti visi simuliacijos įvykiai, simuliacija bus sustabdoma, kai bus įvykdytas įvykis pavadinimu *procesas.*
env.peek(), env – simuliacijos aplinka, *peek()* – metodas pažiūrėti suplanuoto kito įvykio laiką.	Grąžinamas simuliacijos kito įvykio laikas, kada tai įvyks, jeigu nėra jokio kito įvykio, tai yra gaunama rezultate begalybė.
env.step(), env – simuliacijos aplinka, *step()* – metodas pradedantis kitą simuliacijos įvykį.	Pradedamas simuliacijos kitas įvykis, jeigu nėra jokio kito įvykio, tai iškeliama *EmptySchedule* išimtis.

„SimPy“ simuliacijos aplinka turi dvi skirtingas aplinkas, kurios naudojamos skirtingoms sistemoms ir procesams simuliuoti. Šios dvi aplinkos yra: *Environment* ir *RealTimeEnvironment,* detaliau apie jų naudojimą ir skirtumus bus pateikta apačioje:

- *Environment* aplinka naudojama normalioms simuliacijoms, nes šios simuliacijos atliekamas kuo įmanoma greičiau, taip galima simuliuoti paprastas sistemas ir procesus, kur nerūpi labai tikslus laikas. Pavyzdžiai tokių simuliacijų galėtų būti

kaip: bilietų pardavimo simuliavimas, restorano aptarnavimo simuliacija ir kitos simuliacijos, kur nerūpi pats tiksliausias laikas.

- *RealTimeEnvironment* aplinka naudojama tikro laiko simuliacijoms, nes šios simuliacijos atliekamos sinchronizuotai su laikrodžiu laiku. Tokio tipo simuliacijos gali būti naudojamos, kai norima simuliuoti sistemas, kuriose yra visada veikianti aparatūra, žmonių sąveika su simuliacija arba norima analizuoti algoritmo poveikius tikrame laike. Pavyzdys tokios simuliacijos galėtų būti: serverio ir kliento kompiuterio duomenų siuntimo simuliacija.

Tikro laiko simuliacijoms atlikti yra papildomi metodai, kad būtų galima kontroliuoti simuliacijų laiką ir tikslumą. Todėl apačioje bus pateikta lentelė, kurioje bus matomi papildomi metodai ir ką jie daro tikro laiko simuliacijai, tai galima pamatyti 2 lentelėje.

2 lentelė. Tikro laiko simuliacijos aplinkos metodai ir jų poveikis simuliacijai

Tikro laiko simuliacijos aplinkos metodai	**Metodo poveikis simuliacijai.**
Simpy.rt.RealtimeEnvironment(initial_time=x), initial_time=x, kai *x* yra simuliacijos pradinis laikas.	Sukuriama tikro laiko simuliacijos aplinka, kuri pradėta su pradiniu laiku *x*.
Simpy.rt.RealtimeEnvironment(factor=x), factor=x, kai *x* yra simuliacijos laiko faktorius.	Sukuriama tikro laiko simuliacijos aplinka, kurioje simuliacijos laikas keičiasi pagal nustatytą *x* laiko faktorių, t. y. jeigu $x = 60$, tai simuliacijos laiko viena sekundė vyks vieną minutę, o jeigu $x = 0.1$ tai simuliacijos laiko viena sekundė vyks 0.1 sekundės.
Simpy.rt.RealtimeEnvironment(strict=bool), strict=bool, kai *bool* gali būti tik *True* arba *False*.	Sukuriama tikro laiko simuliacijos aplinka, kurioje nustatomas įvykių laiko griežtumas, t. y. jeigu *strict* nustatytas kaip *True*, tai kiekvienas įvykis negali užtrukti ilgiau negu nustatytas laiko faktorius, jeigu

Tikro laiko simuliacijos aplinkos metodai	Metodo poveikis simuliacijai.
	nustatytas kaip *False* tuomet į tai nekreipiamas dėmesys.

Iš šių dviejų aplinkų galima matyti ir nustatyti kokia mums simuliacijos aplinka yra labiau tinkama, nes nėra tikslo kurti tikro laiko simuliacijos aplinkos ir simuliacijos, kai tai sistemai ar procesui neberūpi pats tiksliausias laikas.

Simuliavimo procesai tai yra simuliacijos funkcijos kurios apibrėžia simuliacijos veikimą, „SimPy“ įrankyje šie procesai yra sugeneruojami kaip paprastos Python kalbos kodo funkcijos, kurios turi daugelį simuliacijos įvykių. Kai šie simuliacijos procesai turi bent vieną įvykį jie yra įdedami į simuliavimo aplinką ir tuomet šie procesai laukia, kol jų įvykis bus pradėtas. Užtat kad procesai yra įvykių rinkiniai, todėl bus detaliau kalbama apie simuliacijos įvykius, kuriuose vyksta visas simuliavimo procesas.

Simuliavimo įvykiai yra – atidėjimai, ateities įvykiai ar pažadai, kas reiškia jog simuliacijos įvykis įvyks, šių įvykių stadijos gali būti tokios:

- nutikto (neužfiksuotas),
- nutiks (užfiksuotas) arba
- jau nutiko (apdorotas).

Kiekvienas įvykis praeina kiekvieną iš šių stadijų tik vieną kartą, nuo nenutikusio įvykio iki jau nutikusio įvykio. Simuliacijos visi įvykiai yra surišti laike ir laikas juos valdo, todėl visi įvykiai pereina per visas šias būsenas. Todėl „SimPy“ įrankio simuliacijos įvykiai eina šia eilę: Įvykis neužfiksuotas -> Įvykis užfiksuotas (pridedamas į įvykių eilę simuliacijoje) -> Įvykis apdorotas (išmetamas iš įvykių eilės simuliacijoje), kai įvykis pasiekią užfiksavimo stadiją, tuomet tas simuliacijos įvykis gali būti sustabdoma ir pratęsiamas kada tik reikia, šis funkcionalumas yra prarandamas, kai įvykis būna apdorojamas. Pagrindiniai su įvykiais susiję metodai yra pateikti apačioje 3 lentelėje.

3 lentelė. Simuliacijos įvykių metodai ir jų paaiškinimas

Simuliacijos įvykio metodai	**Metodo paaiškinimas**
yield event, kai *event* yra įvykis.	Sustabdomas simuliacijos įvykis ir pridedamas metodas *_resume()*, kad būtų baigtas įvykis nuo ten kur buvo sustabdytas.
Event.succeed(value=None), value=None yra numatytas grąžinimas, tačiau galima grąžinti ir kitus duomenis po įvykio.	Įvykis įvykdomas sėkmingai ir gaunami duomenys, jeigu jie buvo duodami metode.
Event.fail(exception), exception grąžinimas, kad būtų galima pamatyti, kas nutiko negerai.	Įvykis įvykdomas nesėkmingai ir gaunama informacija apie klaidą, jeigu buvo naudojamas *exception*.
Event.trigger(event), event kitas įvykis ir jo duomenys ir rezultatai.	Pradedamas naujas įvykis ir gaunami duomenys ir rezultatai apie perduotą įvykį, nesvarbu ar įvykis įvyko sėkmingai ar nesėkmingai.
Timeout(delay, value=None), delay įvykio delsimo laikas, *value=None* yra numatytas grąžinimas, tačiau galima grąžinti ir kitus duomenis po įvykio.	Sulaikomas simuliavimas su *Timeout* įvykiu, nauji įvykiai visada įdedami į eile tokiu principu *now + delay*. Pakeičiant *delay* gali būti valdomas laikas, kiek dels šis įvykis simuliacijoje. Bei galima perduoti įvykio duomenis naudojant *value*.
AllOf (arba &) – visi simuliacijos įvykiai.	Tai yra sąlyginis įvykis kurio metu nustatomi įvykiai, kurių turi laukti šis įvykis, kad galėtų pradėti savo darbą, šiuo atveju turi laukti visų įvykių, kurie nustatyti.
AnyOf (arba |) – bent vienas iš visų simuliacijos įvykių.	Tai yra sąlyginis įvykis kurio metu nustatomi įvykiai, kurių turi laukti šis įvykis, kad galėtų pradėti savo darbą, šiuo atveju turi laukti bent vieno iš įvykių, kurie nustatyti.

„SimPy“ įrankyje procesai irgi yra įvykiai, todėl simuliacijos procesai gali būti sustabdyti ir tęsiami simuliacijos naudojant *yield* metodą. Iš lentelės galima matyti jog įvykiai turi daug metodų, kurių pagalba galima sukurti detalią sistemos simuliaciją.

Galiausiai lieka tik simuliacijos bendrai naudojami resursai, kurie „SimPy“ įrankyje yra trijų tipų ir simuliacijos metu procesai ar įvykiai laukia kada jie galės juos panaudoti. Šios trys grupės resursų yra tokios, su paaiškinimais, kur jie naudojami:

- Resursai (angl. *Resources*) – tai tokie resursai, kurie yra limituoja prieigą prie jų procesams tam tikru skaičiumi (klientų aptarnavimo vietos, degalų pylimo vietos, sistemos naudotojų vietos vienu metu).
- Konteineriai (angl. *Containers*) – tai tokie resursai, kurie naudojami produkcijos ir vartojimo modeliavimui, kur resursai vienarūšiai. Šie resursai gali būti nepertraukiami arba atskiri. Šio tipo resursai dažniausiai yra naudojami simuliuoti resursų kiekį (degalinių dujos ir benzinas).
- Saugyklos (angl. *Stores*) – tai tokie resursai, kurie leidžia gaminti ir naudoti Python tipo objektus.

Visi išvardyti resursai turi tą patį principą, tai yra resursai, kuriuos sudaro konteineris, kurio talpa paprastai yra ribota. Simuliacijos procesai gali bandyti įdėti kažką į resursus arba paimti kažką iš resursų, kai resursai yra pilni, kiti procesai turi laukti eilėje, kol jie galės naudoti esamus resursus. Kai įvyksta proceso resursų naudojimo nutraukimas, tuomet kiti procesai turi du pasirinkimus: 1. laukti užklausos, arba 2. gali sustoti laukimo proceso eilėje. Taip galima valdyti kas gali gauti prioritetus resursams, nes tikrose sistemose dažniausiai vieni procesai yra svarbesni negu kiti arba kai kurie procesai turi naudoti resursus tam tikru principu, todėl „SimPy“ turi specializuotas sekas ir įvykius šiems resursams, kaip: prioritetinė eilė ir surūšiuota eilė.

Apačioje pateiktoje 4 lentelėje bus matomi visi esantys resursų tipai „SimPy“ įrankyje ir jų skirtumai:

4 lentelė. Bendrai naudojamų resursų tipai

Bendrai naudojamų resursų tipas	**Bendrai naudojamų resursų paaiškinimas**
Resource(env, capacity=x), env – tai simuliavimo aplinka, *capacity=x* numatytai būna vienas, čia pateikiamas limtas procesams.	Sukuriamas resursas, kurį vienu metu gali pasiekti tiek procesų, kiek nurodyta *x* kintamajame, viskas vyksta eilėje, kas pirmas papuola tas pirmas ir naudoja.
PriorityResource(env, capacity=x), env – tai simuliavimo aplinka, *capacity=x* numatytai būna vienas, čia pateikiamas limtas procesams. *resource.request(priority=y)*, šiuo metodu procesas kreipiasi į resursus su savo prioritetu *y*, kuo skaičius didesnis tuo mažesnis prioritetas.	Sukuriamas resursas, kurį vienu metu gali pasiekti tiek procesų, kiek nurodyti *x* kintamajame, tačiau šie resursai yra paskirstomi pagal gautus prašymus su prioritetu y.
PreemptiveResource(env, capacity=x), env – tai simuliavimo aplinka, *capacity=x* numatytai būna vienas, čia pateikiamas limtas procesams. *resource.request(priotity=y, peempt=Bool)*, šiuo metodu procesas kreipiasi į resursus su savo prioritetu *y*, kuo skaičius didesnis tuo mažesnis prioritetas. Tačiau galima kartu reguliuoti kokie procesai gali užbėgti už akių kitiems procesams, reikšmės *False* (negali) ir *True* (gali).	Sukuriamas resursas, kurį vienu metu gali pasiekti tiek procesų, kiek nurodyti *x* kintamajame, tačiau šie resursai yra paskirstomi pagal gautus prašymus su prioritetu *y*. Prioritetas yra svarbesnis negu už akių užbėgimo vėliava, todėl galima užtikrinti, kad resursai nebandys užeiti už kitų resursų, kurie yra svarbesni.
Container(env, init=x, capacity=y), env – tai simuliavimo aplinka, *init* tai keikis pradinių resursų, *capacity* tai kiekis galimų visų resursų šiame konteineryje.	Sukuriamas konteinerio tipo resursas, kuris gali būti papildytas ir išnaudojamas, todėl šio tipo resursas labiausiai naudojamas ten, kur šis resursas naudojamas ir papildomas.
Store(env, capacity=x), env – tai simuliavimo aplinka, *capacity=x* numatytai būna vienas, čia pateikiamas limtas procesams.	Sukuriamas saugyklos tipo resursas, kuriama saugojami Python objektai arba kitaip sakant daiktai, šis resursas skiriasi nuo kitų tuo, kad

Bendrai naudojamų resursų tipas	**Bendrai naudojamų resursų paaiškinimas**
	šie resursai gali lengviau atvaizduoti kaip naudojami daiktai.
FilterStore(env, capacity=x), env – tai simuliavimo aplinka, *capacity=x* numatytai būna vienas, čia pateikiamas limtas procesams.	Sukuriamas saugyklos tipo resursas, kuris yra filtruojamas automatiškai, kai į jį įdedamas arba išimamas naujas objektas. Šioje saugykloje saugojami Python objektai arba kitaip sakant daiktai, šis resursas skiriasi nuo kitų tuo, kad šie resursai gali lengviau atvaizduoti kaip naudojami daiktai.
PriorityStore(env, capacity=x), env – tai simuliavimo aplinka, *capacity=x* numatytai būna vienas, čia pateikiamas limtas procesams.	Sukuriamas saugyklos tipo resursas, kuris yra prioretizuojamas automatiškai pagal prioritetus, kai į jį įdedamas arba išimamas naujas objektas. Šioje saugykloje saugojami Python objektai arba kitaip sakant daiktai, šis resursas skiriasi nuo kitų tuo, kad šie resursai gali lengviau atvaizduoti kaip naudojami daiktai.

1.3.1. „SimPy“ Diskretinių Įvykių Simuliavimo įrankio analizės išvados

Iš šio įrankio galimybių analizės galima pastebėti jog atvaizduoti sistemos simuliaciją naudojant diskretinių įvykių simuliavimo įrankį galima pakankami lengvai, bei yra daug metodų pasiekti skirtingas simuliacijos versijas pagal pradinę simuliacijos sistemą. Kartu dėl to, kad tai yra atviro kodo diskretinių įvykių simuliavimo įrankis, kuris sukurtas naudojant Python kalbą, šis įrankis gali būti praplėstas naudojant kitas Python bibliotekas, kas padėtų simuliuoti sistemą. Dėl „SimPy“ įrankio galimybių šis įrankis buvo naudojamas simuliuoti daugelį skirtingų sistemų, kaip antro sandėlio atidarymo simuliacijai, kurios metu buvo simuliuojami surinkti duomenys apie atliktus drabužių nuomas, pirkimus ir gražinimus, kas padėjo gauti svarbią informaciją apie antro sandėlio poveikį pirmajam sandėliui, bei klientams, kurie gyvena tolesnėse Amerikos valstijose lyginant su pirmuoju sandėliu [Hei19]. Arba miško veiklos simuliavimui, kur buvo simuliuojami

įvairūs miško veiklos tiekimo grandinės scenarijai. Šių simuliacijų rezultatais buvo galima įvertinti vėlavimus ir gedimus kasdieniniuose darbo planuose [PCB16]. Šie du darbai patvirtina, kad atviro kodo „SimPy“ diskretinių įvykių simuliavimo įrankis gali atnešti naudos įmonėmis simuliuojant jų procesus.

1.4. Analitinės dalies išvados

Atlikus visų mokslinių šaltinių ir kitų elektroninių šaltinių analizę buvo pamatyta ankstyvų prototipų nauda sistemos kokybei patikrinti, bei jų pagalba buvo galima nustatyti problemas ir išvengiamos šių problemų. Tuo pačiu buvo matoma prototipų nauda ne tik informacijos technologijų srityje, bet ir kitose srityse, kur norint įsitikinti sistemų veikimą, virtualūs prototipai buvo vienas iš saugiausių būdų tai atlikti. Virtualūs prototipai buvo naudojami automobilių ir roboto technikos sistemoms patikrinti ir įvertinti. Tuomet kitame skyriuje buvo analizuojami diskretinių įvykių simuliavimo įrankių privalumai ir pačių diskretinių įvykių simuliavimo nauda kompanijoms. Pagrindiniai pastebėjimai buvo tokie, jog simuliuojamos sistemos šiais įrankiais gauna papildomus duomenis apie savo sistemų veikimą, bei sujungiant su kitais duomenimis galima sukurti modelius, kurie optimizuotų simuliuojamas sistemas. Tačiau pagrindinis privalumas šių įrankių yra tai jog sistemų simuliavimo metu naudojant diskretinių įvykių simuliavimo įrankius buvo galima numatyti tam tikras problemas arba buvo gaunami duomenys, kurių dėka buvo galima priimti geresnius svarbiausius sprendimus. Galiausiai buvo analizuojami atviro kodo ir komerciniai diskretinių įvykių simuliavimo įrankiai, kurių metu buvo pamatyta jog atviro kodo įrankiai gali prilygti komerciniams. Tuo pačiu atviro kodo įrankiai leidžia mažoms ir vidutinio dydžio kompanijoms išbandyti šiuos įrankius sistemos supratimui, optimizavimui ar sprendimų parėmimui. Kartu buvo analizuojamas atviro kodo „SimPy“ diskretinių įvykių simuliavimo įrankis, kuris naudojamas šio magistrinio darbo praktiniai daliai atlikti. Kiuramai sistemos prototipo simuliacijos analizei bus naudojami „SimPy“ diskretinių įvykių simuliavimo įrankio simuliacijos metu gautų duomenų log failai, kurie bus sujungti su Python kalbos „Matplotlib“ biblioteka. Naudojant „Matplotlib“ biblioteką bus sukurti grafinės diagramos ir grafiškai atvaizduoti simuliacijos duomenys, kas padės analizuoti sistemos simuliaciją ir priimti simuliacijos pakeitimų sprendimus. Šios analizės metu buvo aiškinamasi kaip veikia „SimPy“ įrankis ir kaip sistemos gali būti simuliuojamos naudojant jį [Hei19] ir [PCB16]. Šio įrankio pagalba buvo galima numatyti ir pasiruošti antro sandėlio plėtimusi, miško veiklos klaidų ir darbo

vėlavimų įvertinimui. Todėl atviro kodo „SimPy“ diskretinių įvykių simuliavimo įrankis bus naudojamas kartu su „Matplotlib“ Python biblioteka atlikti praktiniai daliai ir optimizuoti kelių sankryžos prototipą.

2. Praktinė dalis

2.1. Projektavimas

Šio darbo simuliacijos prototipo kūrimui buvo pasirinkta sukurti transporto priemonių ir pėsčiųjų sankryžos prototipas. Būtina paminėti, kad šis prototipas apima šiuos atskirus dalykus, kurie sudaro pilną simuliacijos aplinkos prototipą:

1. Transporto priemonių maršrutas – simuliacijos aplinkoje tai būtų maršrutas, kuriuo gali keliauti aplinkos elementai, kurie priklauso Transporto priemonė grupei.
2. Pėsčiųjų perėja – simuliacijos aplinkoje tai būtų perėja, kuria gali keliauti aplinkos elementai, kurie priklauso Pėsčiasis grupei.
3. Transporto priemonė – simuliacijos aplinkoje tai būtų objektas, kuris keliauja aplinkos elementu Transporto priemonių maršrutais.
4. Pėsčiasis – simuliacijos aplinkoje tai būtų objektas, kuris keliauja aplinkos elementu Pėsčiųjų perėjomis.
5. Šviesoforas – simuliacijos aplinkoje tai būtų objektas, kuris valdo sankryžos taisykles.

Sudėjus visus šiuos skirtingus prototipo elementus galima sukurti skirtingus kelių sankryžos prototipus. Detaliau apie visus simuliacijos aplinkos elementus („Transporto priemonių maršrutas“, „Pėsčiųjų perėja“ ir kt. elementai) ir kaip jie veikia bus aptarta objekto realizacijos skyriuje.

Transporto priemonių ir pėsčiųjų sankryžos prototipui sukurti buvo pasirinktas „SimPy“ biblioteka, kuri yra diskretinių įvykių simuliacijos įrankis, šio įrankio pagalba visi išvardyti prototipo elementai gali būti sukurti naudojant „Python“ kalbos principus. Tuo pačiu, kad prototipas veiktų kaip priklauso, turi būti sukurti transporto priemonių ir pėsčiųjų generatoriai, kurių pagalba galima sukurti atvykusias transporto priemones ir pėsčiuosius sankryžoje. Šie sukurti objektai gali būti traktuojami kaip „SimPy“ simuliacijos aplinkos įvykiai (angl. *events*), kurie sąveikautų su kitais simuliacijos aplinkos įvykiais kaip: transporto priemonių maršrutai, pėsčiųjų perėjos ir šviesoforas.

Norint įsitikinti, jog transporto priemonių ir pėsčiųjų generatoriai sukurtų naujus objektus nepriklausomai viena nuo kito su tam tikru vidurkiu, turi būti naudojamas Puasono pasiskirstymas

[Sci22]. Tam pasiekti galima naudoti „NumPy“ bibliotekos funkcijas [Num22] kaip: *numpy.random.poisson()* arba *numpy.random.exponential()*.

Galiausiai norint surinkti statistinius duomenis, kurie gaunami simuliacijos aplinkos metu reikia sukurti objektą, kurio tikslas būtų surinkti kiekvieno maršruto, ir perėjos laukimo laiką bei vidurkį, kad būtų galima tai atvaizduoti naudojant „MatplotLib“ biblioteka [MDT22], kurios pagalba galima braižyti skirtingas diagramas „Pyhton“ kodo aplinkoje.

2.2. Realizacija

Simuliacijos prototipui buvo sukurta dviejų kelių susikirtimo sankryža. Tokia sankryža buvo pasirinkta, nes ji tokio pat tipo kaip didžioji dalis sankryžų t. y. susikerta du keliai iš kurių yra vienas pagrindinis, o kitas papildomas kelias, kuris kertasi per sankryžą. Kelio sankryžą sudaro iš viso 4 kelios dalys, kiekviena kelio dalis turi po vieną transporto priemonių liniją ir ši sankryža yra reguliuojama šviesos šviesoforų. Žinoma dėl to jog yra šviesoforai, tai reiškia jog gali pėstieji pereiti per kelią vadovaujantis šviesoforo būsena. Galiausiai kiekviena transporto priemonių linija turi galimybę važiuoti tiesiai, važiuoti į kairę arba važiuoti į dešinę ir nėra jokių papildomų kelio ženklų ar kelio žymių. Todėl ši sankryža buvo pasirinkta kaip pagrindas pagal kurią būtų sukurta simuliacijos aplinka ir modeliuojamas visas prototipas.

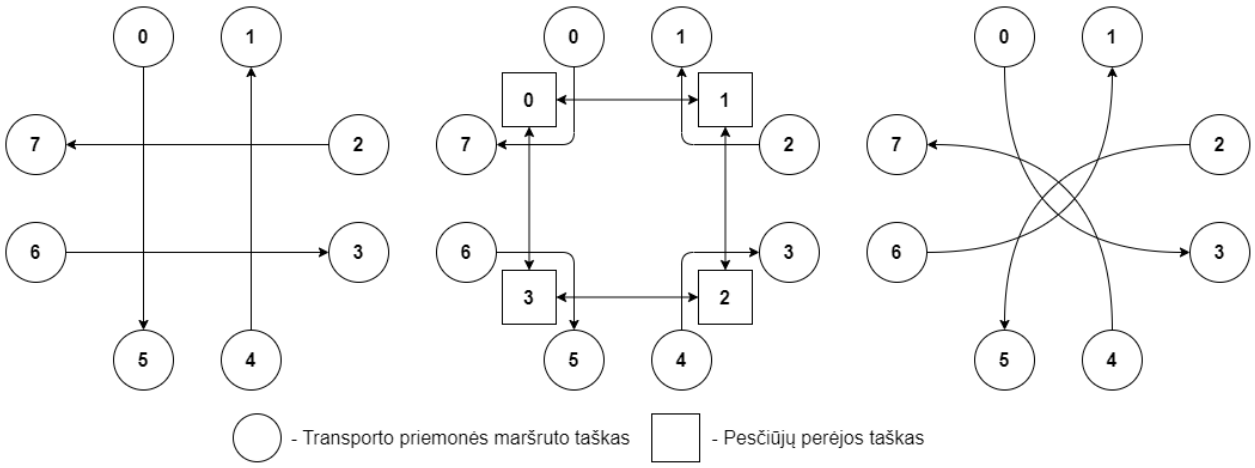
Pasirinktos sankryžos modeliavimui buvo nuspręsta sukurti šios sankryžos kelių tinklo modelį naudojant nukreipiamą grafą, kuris būtų sudarytas iš dviejų elementų ***G = (V, A)***:

- V, tai taškų rinkinys;
- A, tai nukreiptų kraštų rinkinys.

Naudojant šį nukreipiamą grafą galima sukurti bet kokį sankryžos prototipą ir kartu galima nurodyti iš kurio grafo taško galima pasiekti kokį grafo tašką naudojant nukreipiamuosius kraštus. Kuriamos sankryžos nukreipiamo grafiko diagrama pateikta **1** pav. Sankryžos nukreipiamo grafo diagrama. Naudojantis šia diagrama galima matyti kaip sankryžoje susijungia visi jai priklausantys elementai:

- Transporto priemonės maršruto taškai, kurie pažymėti apskritimu ir turi savo Id;
- Pėsčiųjų perėjos taškai, kurie pažymėti kvadratu ir turi savo Id;
- Transporto priemonės galimi posūkiai, kurie susijungia tarp dviejų transporto priemonės maršruto taškų;

- Pėsčiųjų perėjos galimi perėjimai, kurie susijungia tarp dviejų pėsčiųjų perėjos taškų.



1 pav. Sankryžos nukreipiamo grafo diagrama

Kad diagramą būtų įskaitoma ir lengviau suprantama kiekvienas posūkis yra atskirtas vienas nuo kito. Pirmoji diagramos dalis kairėje nurodo iš kokių taškų į kuriuos taškus transporto priemonės gali važiuoti transporto priemonės tiesiai, vidurinė diagramos dalis nurodo iš kokių taškų į kuriuos taškus transporto priemonės gali važiuoti į dešinę ir dešinė diagramos dalis nurodo iš kokių taškų į kuriuos taškus transporto priemonės gali važiuoti į kairę. Tuo pačiu vidurinėje diagramos dalyje yra pateikti pėsčiųjų kelių taškai ir jų susijungimai, tik reikia atkreipti dėmesį, jog pėsčiųjų perėjų taškai yra sujungti dvipusiu ryšių, t. y. jog pėstieji gali eiti pirmyn ir atgal tarp dviejų sujungtų taškų. Galiausiai būtina paminėti, jog ši sankryžą yra valdoma šviesoforo, kuris nėra pažymėtas diagramoje, nes jis nėra būtinas norint sukurti nukreipiamąjį grafą simuliacijai sukurti. Apačioje 5 lentelėje yra pateikta kokie kintamieji yra gaunami ir sukuriami naudojantis šia diagrama.

5 lentelė. Nukreipiamo grafo diagramos pagalba sukurti kintamieji

Kintamieji	**Paaiškinimas**
Nodes = [0, 1, 2, 3, 4, 5, 6, 7]	Nukreipiamojo grafo taškų rinkinys, kurio pagalba žinoma kokie taškai yra grafe.
Links = { *0: [5, 7, 3],*	Nukreipiamojo grafo nukreiptų kraštų rinkinys, šio rinkinio pagalba galima atvaizduoti iš kurio taško galima važiuoti į kurį tašką. Pavyzdžiui matome, jog grafo

Kintamieji	**Paaiškinimas**
2: [7, 1, 5], *4: [1, 3, 7],* *6: [3, 5, 1]* *}*	taškas 0 (transporto priemonių kelio taškas 0) yra susietas su trejais kitais grafo taškais 5, 7 ir 3. Taip žinome jog egzistuoja tokie sujungimai: (0, 5), (0, 7) ir (0, 3) iš taško 0. Naudojama sukurti transporto priemonių maršrutus, klasės *Road*.
LinkGroups = [0, 1, 0, 1]	Šis sąrašas parodo, kokie pirminiai taškai prikalusio kuriai šviesoforo grupei. Atsižvelgiant į sankryžos diagramą ir *Links* rinkinį galime matyti jog 0 ir 4 taškai yra 0 grupės, o 2 ir 6 taškai yra 1 grupės taškai. Naudojama *TrafficLight* klasės.
Road_Weights = [0.35, 0.15, 0.35, 0.15]	Šio sąrašo pagalba galima nurodyti, kokie transporto priemonių maršrutų taškai turi didesnį transporto priemonių srautą (atvyksta daugiau transporto priemonių). Dabartiniai parametrai parodo jog maršrutų taškai 0 ir 4 turi po 35% visų transporto priemonių srauto, o 2 ir 6 taškai turi tik po 15% transporto priemonių srauto. Naudojama *carGenerator* klasės.
Destination_Weights = [0.40, 0.40, 0.20]	Šio sąrašo pagalba galima nurodyti, kur dažniausiai važiuoja transporto priemonės. Dabartiniai parametrai parodo jog tiesiai važiuoja 40%, į dešinę 40% ir į kairę 20% transporto priemonių. Naudojama *carGenerator* klasės.
PedNodes = [0, 1, 2, 3]	Nukreipiamojo grafo taškų rinkinys, kurio pagalba žinoma kokie taškai yra grafe.
PedLinks = [*[0, 1],* *[1, 0],*	Nukreipiamojo grafo nukreiptų kraštų rinkinys, šio rinkinio pagalba galima atvaizduoti iš kurio taško galima eiti į kurį tašką. Dėl to jog pėsčiųjų perėjos turi viena baigiamąjį tašką iš pradžios taško, visi kelionės taškai atvaizduoti kaip sąryšis tarp dviejų taškų. Tuo pačiu

Kintamieji	**Paaiškinimas**
[1, 2], *[2, 1],* *[2, 3],* *[3, 2],* *[3, 0],* *[0, 3]* *]*	pėstieji gali eiti ta pačia pėsčiųjų perėja iš vienos pusės ir kitos pusės, todėl pasikartoja ta pati perėja, bet iš skirtingų kelio pusių. Pavyzdžiui [0, 1] ir [1, 0] yra ta pati pėsčiųjų perėja, tačiau pradžios ir kelionės taškai yra skirtingi. Naudojama sukurti pėsčiųjų perėjas, klasės *Path*.
PedLinkGroups = [1, 1, 0, 0, 1, 1, 0, 0]	Šis sąrašas parodo, kokie pėsčiųjų perėjos sujungimai tarp taškų prikalusio kuriai šviesoforo grupei. Atsižvelgiant į sankryžos diagramą ir Ped*Links* rinkinį galime matyti jog [0, 1], [1, 0], [2, 3] ir [3, 2] priklauso 1 grupei, o [1, 2], [2, 1], [3, 0] ir [0, 3] priklauso 0 grupei. Naudojama *TrafficLight* klasės.
Path_Weights = [0.125, 0.125, 0.125, 0.125, 0.125, 0.125, 0.125, 0.125]	Šio sąrašo pagalba galima nurodyti, kokie pėsčiųjų perėjos taškų sujungimai kokį turi šansą jog atvyks pėsčiasis. Dabartiniai parametrai jog visos perėjos turi vienoda šansą. Naudojama *pedGenerator* klasės

2.2.1. Simuliacijos aplinka

Simuliacijos aplinkai sukurti yra naudojama „SimPy“ biblioteka, šioje bibliotekoje pats svarbiausias dalykas yra ***simpy.Environment()*** arba ***simpy.rt.RealtimeEnvironment(factor=x)***. Šios dvi funkcijos sukuria simuliacijos aplinką, kuri sąveikauja su visais kitais „SimPy“ elementais. Todėl visos klasės, kurios egzistuoja simuliacijos aplinkoje, turi gauti objektą ***env***. Šio objekto pagalba sukurta klasė žino, kad ji yra simuliacijos aplinkoje ir gali bendrauti ir valdyti kitus simuliacijos aplinkos elementus naudojant „Python“ kalbos sąlygas arba „SimPy“ bibliotekos funkcijas ir įvykių valdymo funkcijas.

Sankryžos simuliacijos aplinkai sukurti buvo naudojamos abi „SimPy“ aplinkos. Tikro laiko simuliacijos pagalba galima lengviau suprasti kaip aplinkoje vyksta skirtingi objektų įvykiai, nes

tikro laiko simuliacijos aplinkoje galima nustatyti kiek laiko trunka simuliacijoje viena sekundė naudojant ***factor=x*** kintamąjį. Šios simuliacijos aplinkos pagalba buvo tikrinama ar viskas veikia kaip priklauso. O galutiniai variantui yra naudojama įprasta simuliacijos aplinka, kurioje viskas įvyksta kuo įmanoma greičiau, todėl norint simuliuoti, kas vyksta visą valandą, nereikia laukti visos valandos.

Apačioje 6 lentelėje yra pateikta kokie kintamieji buvo sukurti, kad veiktų „SimPy“ simuliacijos aplinka.

6 lentelė. Simuliacijos aplinkos veikimui sukurti kintamieji ir metodai

Kintamieji/Metodai	**Paaiškinimas**
RANDOM_SEED = 10	Kintamasis, kurio pagalba galima atkurti rezultatus.
random.seed(RANDOM_SEED)	„NumPy“ bibliotekos funkcija, kurios pagalba atsitiktiniai įvykiai yra priklausomi nuo kintamojo *RANDOM_SEED*, taip galima atkurti tokius pačius rezultatus.
SIMULATION_TIME = 3600	Kintamasis, kuris parodo kiek turi vykti simuliacija. Simuliacijos laikas 3600s. = 1val.
env = simpy.rt.RealtimeEnvironment(factor=0.1)	Tikro laiko simuliacijos aplinkos objekto kūrimas. Šios simuliacijos metu per 1 sekundę įvyksta 10 sekundžių, dėl *factor=0.1* kintamojo,
env = simpy.Environment()	Simuliacijos aplinkos objekto kūrimas. Pagrindinis simuliacijos objektas, kurio pagalba simuliacija gali vykti ir būti valdoma.
env.run(until=SIMULATION_TIME)	Simuliacijos aplinkos paleidimo funkcija iki nurodyto laiko, kuris yra *until=SIMULATION_TIME.*

2.2.2. Simuliacijos klasė Road

Sukurti sankryžos simuliacijos prototipui reikia pirmiausia sukurti transporto priemonių maršrutus, tam padaryti buvo sukurta „Python“ kalbos klasė pavadinimu ***Road***. Šios klasės pagrindinis funkcionalumas yra sukurti maršrutą, kuris turi transporto priemonių seką ir prižiūri ar transporto priemonė gali išvažiuoti į sankryžą ar ne, pagal šviesoforo būsena ir „SimPy“ bibliotekos įvykių paleidimus. Šios klasės kintamieji yra pateikti 7 lentelėje ir klasės „Python“ kalbos kodas pateiktas 2 paveikslėlyje. Žemiau detaliau aprašyta ką atlieka ši klasė ir ką daro kiekvienas jos esantis metodas.

Klasę ***Road*** sudaro šie metodai:

- def __init__(self, env, start, ends) – šis metodas yra konstruktorius, kuris gauna 3 įvestis, kad būtų galima sukurti šios klasės objektą. Pati pirma įvestis yra *env*, tai yra simuliacijos aplinkos objektas sukurtas naudojant „SimPy“ metodą *simpy.Environment(),* tuomet duodamas pradinis transporto priemonės maršruto taškas pavadinimu *start* ir galimos šio maršruto pabaigos (posūkiai, sakykime tiesiai, į kairę ir į dešinę) pavadinimu *ends*. Įvestis *start* ir *ends* gaunami iš sudaryto nukreipiamojo grafo elementų pavadinimu *Links*. Konstruktoriaus viduje sukuriamas vidinis ID, simuliacijos aplinka, kuriai priklauso šis objektas, maršruto pradžia, maršruto pabaigos, transporto priemonių eilė naudojant *deque()* metodą iš „Collections“ bibliotekos, maršruto grupės elementas, ar galima važiuoti vėliava ir du įvykiai žaliai ir raudonai šviesai naudojant „SimPy“ bibliotekos metodą *env.event().*
- def setGroup(self, group) – šis metodas naudojamas priskirti šiam maršrutui šviesoforo šviesos grupę.
- def greenLight(self) – šis metodas naudojamas leisti šio maršruto transporto priemonėms važiuoti per žalią šviesą. Metodo viduje iškviečiamas žalios šviesos įvykio paleidiklis pavadinimu *self.greenLightTrigger.succeed()* naudojant „SimPy“ metodą *succed()*. Taip simuliacijos aplinka žino, jog būtent šiame maršrute šviečia žalia šviesa ir leidžiama šio maršruto transporto priemonėms važiuoti. Tuomet sukuriamas naujas įvykio paleidiklis tuo pačiu pavadinimu *self.greenLightTrigger = self.env.event()*, kad šis maršrutas būtų pasiruošęs, kitam žalios šviesos įvykio iškvietimui. Kartu pakeičiama vidinė vėliava į *self.drivable =*

True, kad naujai sukurtos transporto priemonės, kurios nespėjo išgirsti įvykio paleidiklio *self.greenLightTrigger* galėtų važiuoti, kol šviečia žalia šviesa, nes kitaip naujos transporto priemonės šiame maršrute lauktų kitos žalios šviesos, nors dabar šviečia žalia. Galiausiai jeigu yra bent viena transporto priemonė eilėje leidžiama joms važiuoti sukuriant naują simuliacijos aplinkos procesą ir iškviečiant metodą *env.process(self.letCarsDepart())*.

- def redLight(self) – šis metodas naudojamas neleisti šio maršruto transporto priemonėms važiuoti per raudoną šviesą. Metodo viduje iškviečiamas raudonos šviesos įvykio paleidiklis pavadinimu *self.redLightTrigger.succeed()* naudojant „SimPy“ metodą *succeed()*. Taip simuliacijos aplinka žino, jog būtent šiame maršrute šviečia raudona šviesa ir neleidžiama šio maršruto transporto priemonėms važiuoti. Tuomet sukuriamas naujas įvykio paleidiklis tuo pačiu pavadinimu *self.redLightTrigger= self.env.event()*, kad šis maršrutas būtų pasiruošęs, kitam raudonos šviesos įvykio iškvietimui. Galiausiai pakeičiama vidinė vėliava į *self.drivable = False*, kad naujai sukurtos transporto priemonės, kurios nespėjo išgirsti įvykio paleidiklio *self.redLightTrigger* negalėtų važiuoti, kol šviečia raudona šviesa, nes kitaip naujos transporto priemonės šiame maršrute bandytų važiuoti, nors dabar šviečia raudona.
- def addCar(self, car) – šis metodas naudojamas pridėti transporto priemones į šio maršruto transporto priemonių sąrašą, kuris veikia FIFO (angl. First In, First Out) principu. Kai duodama transporto priemonė, kuri atsirado šiame maršrute tikrinama maršruto vėliava *self.drivable == False* ar negalima važiuoti arba ar yra transporto priemonių eilė *len(self.cars)*. Jeigu ši sąlyga tenkinama tai transporto priemonės objektas yra pridedamas į šio maršruto objekto transporto priemonių sąrašą naudojant šį metodą *self.cars.append(car)*. Tuomet sukuriamas simuliacijos aplinkos procesas ir iškviečiamas transporto priemonės važiavimo metodas *env.process(car.drive())*. Galiausiai jeigu sąlyga nėra tenkinama, kitaip sakant nėra transporto priemonių eilės arba leidžiama važiuoti, tai transporto priemonės nereikia dėti į šio maršruto transporto priemonių sąrašą ir galima sukurti šios transporto priemonės simuliacijos aplinkos procesą ir iškviesti transporto priemonės važiavimo metodą *env.process(car.drive())*.

- def letCarsDepart(self) – šis metodas naudojamas leisti visoms šio maršruto transporto priemonių sąraše esančioms transporto priemonėms važiuoti. Metodo viduje vyksta ciklas, kuris vyksta tol kol šiame maršrute galima važiuoti ir šio maršruto transporto priemonių sąraše yra bent 1 transporto priemonė. Kai ši sąlyga tenkinama, pati pirma transporto priemonė yra išmetama iš šio maršruto transporto priemonių sąrašo naudojant FIFO principą ir iškviečiamas simuliacijos aplinkos įvykis būtent šiai transporto priemonei, kad ji simuliacijos aplinkoje gali užbaigti savo įvykį (važiavimą) naudojant metodą *car.canDrive.succeed()*. Galiausiai šis metodas vyksta simuliacijos aplinkos viduje, nes buvo iškviestas kaip „SimPy“ procesas, todėl palaukia 1.5s., kad kita transporto priemonė privažiuotų iki kelio sankryžos pradžios naudojant metodą *yield env.timeout(1.5)*.

7 lentelė. Klasės Road sukurti kintamieji

Kintamieji	**Paaiškinimas**
roads = [Road(env, start, ends) for start, ends in Links.items()]	Sąrašas *roads* laiko visus sukurtus transporto priemonių maršrutus, maršrutai yra sukuriami naudojant *Links*, kur laikomi visų transporto priemonių maršrutų nukreipiami kraštai. Būtinai duodamas simuliacijos aplinkos objektas *env*.

```
class Road:
    idCounter = 0

    def __init__(self, env, start, ends):
        self.id = "Road_" + str(Road.idCounter)
        Road.idCounter += 1

        self.env = env
        self.start = start
        self.ends = ends
        self.cars = deque()
        self.group = 0
        self.drivable = False
        self.greenLightTrigger = env.event()
        self.redLightTrigger = env.event()

    def setGroup(self, group):
        self.group = group

    def greenLight(self):
```

```
        print("Mašinų kelias ", self.start, " mašinos gali važiuoti")
        self.greenLightTrigger.succeed()
        self.greenLightTrigger = self.env.event()
        self.drivable = True

        if len(self.cars) > 0:
            env.process(self.letCarsDepart())

    def redLight(self):
        print("Mašinu kelias ", self.start, " mašinos negali važiuoti")
        self.redLightTrigger.succeed()
        self.redLightTrigger = self.env.event()
        self.drivable = False

    def addCar(self, car):
        if self.drivable == False or len(self.cars):
            self.cars.append(car)
            env.process(car.drive())
        else:
            env.process(car.drive())

    def letCarsDepart(self):
        # Užtat kaip veikia SimPy turi būti bet koks yield
        while self.drivable == True and len(self.cars) > 0:
            car = self.cars.popleft()
            print("Mašina ", car.id, " išvyko šiuo laiku %.3f, palikdama %d
mašinu eilėje iš Mašinu kelio " % (env.now, len(self.cars)), self.start)

            car.canDrive.succeed()

            # Po pirmos mašinos eilėje esantis delay, kad kita mašina būtų
pirma
            yield env.timeout(1.5)
```

2 pav. Klasės Road kodas

2.2.3. Simuliacijos klasė Path

Sukurti sankryžos simuliacijos prototipui reikia sukurti pėsčiųjų perėjas, tam padaryti buvo sukurta „Python“ kalbos klasė pavadinimu ***Path***. Šios klasės pagrindinis funkcionalumas yra sukurti perėją, kuri turi pėsčiųjų seką ir prižiūri ar pėsčias gali išeiti į sankryžą ar ne, pagal šviesoforo būsena ir „SimPy“ bibliotekos įvykių paleidimus. Ši klasė veikia panašiai kaip ***Road*** klasė, tačiau turi keletą skirtumų: pėsčiųjų perėjos turi tik vieną pabaigos tašką ir eilėje esantys pėstieji neturi laukti kitų pėsčiųjų eilėje, eina kada gali. Šios klasės kintamieji yra pateikti 8 lentelėje ir klasės „Python“ kalbos kodas pateiktas 3 paveikslėlyje. Žemiau detaliau aprašyta ką atlieka ši klasė ir ką daro kiekvienas jos esantis metodas.

Klasę ***Road*** sudaro šie metodai:

- def __init__(self, env, start, end) – šis metodas yra konstruktorius, kuris gauna 3 įvestis, kad būtų galima sukurti šios klasės objektą. Pati pirma įvestis yra *env*, tai yra simuliacijos aplinkos objektas sukurtas naudojant „SimPy“ metodą *simpy.Environment()*, tuomet duodamas pėsčiųjų perėjos pradžios taškas pavadinimu *start* ir pėsčiųjų perėjos pabaigos taškas pavadinimu *end*. Įvestis *start* ir *end* gaunami iš sudaryto nukreipiamojo grafo elementų pavadinimu *PedLinks*. Konstruktoriaus viduje sukuriamas vidinis ID, simuliacijos aplinka, kuriai priklauso šis objektas, pėsčiųjų perėjos pradžia, pėsčiųjų perėjos pabaiga, pėsčiųjų eilė naudojant *deque()* metodą iš „Collections“ bibliotekos, pėsčiųjų perėjos grupės elementas, ar galima eiti vėliava ir du įvykiai žaliai ir raudonai šviesai naudojant „SimPy“ bibliotekos metodą *env.event()*.
- def setGroup(self, group) – šis metodas naudojamas priskirti šiai pėsčiųjų perėjai šviesoforo šviesos grupę.
- def greenLight(self) – šis metodas naudojamas leisti šios pėsčiųjų perėjos pėstiesiems eiti per žalią šviesą. Metodo viduje iškviečiamas žalios šviesos įvykio paleidiklis pavadinimu *self.greenLightTrigger.succeed()* naudojant „SimPy“ metodą *succeed()*. Taip simuliacijos aplinka žino, jog būtent šioje pėsčiųjų perėjoje šviečia žalia šviesa ir leidžiama šios perėjos pėstiesiems eiti. Tuomet sukuriamas naujas įvykio paleidiklis tuo pačiu pavadinimu *self.greenLightTrigger = self.env.event()*, kad ši pėsčiųjų perėja būtų pasiruošusi, kitam žalios šviesos įvykio iškvietimui. Kartu pakeičiama vidinė vėliava į *self.walkable = True,* kad naujai sukurti pėstieji, kurie nespėjo išgirsti įvykio paleidiklio *self.greenLightTrigger* galėtų eiti, kol šviečia žalia šviesa, nes kitaip nauji pėstieji šioje pėsčiųjų perėjoje lauktų kitos žalios šviesos, nors dabar šviečia žalia. Galiausiai jeigu yra bent vienas pėsčiasis eilėje leidžiama jiems eiti sukuriant naują simuliacijos aplinkos procesą ir iškviečiant metodą *env.process(self.letPedsDepart()).*
- def redLight(self) – šis metodas naudojamas neleisti šios pėsčiųjų perėjos pėstiesiems eiti per raudoną šviesą. Metodo viduje iškviečiamas raudonos šviesos įvykio paleidiklis pavadinimu *self.redLightTrigger.succeed()* naudojant „SimPy“ metodą *succeed()*. Taip simuliacijos aplinka žino, jog būtent šioje pėsčiųjų perėjoje šviečia raudona šviesa ir neleidžiama šios pėsčiųjų perėjos pėstiesiems eiti. Tuomet sukuriamas naujas įvykio paleidiklis tuo pačiu pavadinimu *self.redLightTrigger=*

self.env.event(), kad ši pėsčiųjų perėja būtų pasiruošusi, kitam raudonos šviesos įvykio iškvietimui. Galiausiai pakeičiama vidinė vėliava į *self.walkable = False*, kad naujai sukurti pėstieji, kurie nespėjo išgirsti įvykio paleidiklio *self.redLightTrigger* negalėtų eiti, kol šviečia raudona šviesa, nes kitaip nauji pėstieji šioje pėsčiųjų perėjoje bandytų eiti, nors dabar šviečia raudona.

- def addPed(self, ped) – šis metodas naudojamas pridėti pėsčiuosius į šios pėsčiųjų perėjos pėsčiųjų sąrašą. Kai duodamas pėsčiasis, kuris atsirado šioje pėsčiųjų perėjoje, tikrinama pėsčiųjų perėjos vėliava *self.walkable == False* ar negalima eiti arba ar yra pėsčiųjų eilė *len(self.peds)*. Jeigu ši sąlyga tenkinama tai pėsčiojo objektas yra pridedamas į šios pėsčiųjų perėjos objekto pėsčiųjų sąrašą naudojant šį metodą *self.peds.append(ped)*. Tuomet sukuriamas simuliacijos aplinkos procesas ir iškviečiamas pėsčiojo ėjimo metodas *env.process(ped.walk())*. Galiausiai jeigu sąlyga nėra tenkinama, kitaip sakant nėra pėsčiųjų eilės arba leidžiama eiti, tai pėsčiojo nereikia dėti į šios pėsčiųjų perėjos pėsčiųjų sąrašą ir galima sukurti šio pėsčiojo simuliacijos aplinkos procesą ir iškviesti pėsčiojo ėjimo metodą *env.process(ped.walk())*.
- def letPedsDepart(self) – šis metodas naudojamas leisti visiems šios pėsčiųjų perėjos pėsčiųjų sąraše esantiems pėstiesiems eiti. Metodo viduje vyksta ciklas, kuris vyksta tol kol šioje pėsčiųjų perėjoje galima eiti ir šios pėsčiųjų perėjos pėsčiųjų sąraše yra bent 1 pėsčiasis. Kai ši sąlyga tenkinama, pats pirmas pėsčiasis yra išmetamas iš šios pėsčiųjų perėjos pėsčiųjų sąrašo ir iškviečiamas simuliacijos aplinkos įvykis būtent šiam pėsčiajam, kad jis simuliacijos aplinkoje gali užbaigti savo įvykį (ėjimą) naudojant metodą *ped.canWalk.succeed()*. Galiausiai šis metodas vyksta simuliacijos aplinkos viduje, nes buvo iškviestas kaip „SimPy“ procesas, todėl turi palaukti 0.0s., kad šis procesas galėtų veikti be klaidų naudojant metodą *yield env.timeout(0.0)*.

8 lentelė. Klasės Path sukurti kintamieji

Kintamieji	**Paaiškinimas**
paths = [Path(env, start, end) for start, end in PedLinks]	Sąrašas *paths* laiko visas sukurtas pėsčiųjų perėjas, perėjos yra sukuriamos naudojant *PedLinks*, kur laikomi visų pėsčiųjų perėjų nukreipiami kraštai. Būtinai duodamas simuliacijos aplinkos objektas *env*.

```
class Path:
    idCounter = 0

    def __init__(self, env, start, end):
        self.id = "Path_" + str(Path.idCounter)
        Path.idCounter += 1

        self.env = env
        self.start = start
        self.end = end
        self.peds = deque()
        self.group = 0
        self.walkable = False
        self.greenLightTrigger = env.event()
        self.redLightTrigger = env.event()

    def setGroup(self, group):
        self.group = group

    def greenLight(self):
        print("Pesciuju Pradzios Taskas ", self.start, " pestieji gali eiti
i ", self.end)
        self.greenLightTrigger.succeed()
        self.greenLightTrigger = self.env.event()
        self.walkable = True

        if len(self.peds) > 0:
            env.process(self.letPedsDepart())

    def redLight(self):
        print("Pesciuju pradzios taskas ", self.start, " pestieji negali
eiti i ", self.end)
        self.redLightTrigger.succeed()
        self.redLightTrigger = self.env.event()
        self.walkable = False

    def addPed(self, ped):
        if self.walkable == False or len(self.peds):
            self.peds.append(ped)
            env.process(ped.walk())
        else:
```

```
            env.process(ped.walk())

    def letPedsDepart(self):
        # Užtat kaip veikia SimPy turi būti bet koks yield
        while self.walkable == True and len(self.peds) > 0:
            ped = self.peds.popleft()
            print("Pestysis  ", ped.id, " pradejo eiti siuo laiku %.3f,
palikdamas %d pesciuju eileje is Pesciuju Kelio tasko " % (env.now,
len(self.peds)), self.start)

            ped.canWalk.succeed()

            # Dėl to kaip veikia pėsčiųjų perėjos, nereikia laukti pirmo
pėsčiojo
            yield env.timeout(0.0)
```

3 pav. Klases Path kodas

2.2.4. Simuliacijos klasė Car

Sankryžos simuliacijos aplinkos prototipui vienas iš svarbiausių elementų būtų pati transporto priemonė, kad būtų galima sekti skirtingus duomenis apie šį elementą, kartu reikia kaupti papildoma informaciją kaip: kiek užtruko laiko kelionė ar kiek užtruko laiko laukti transporto priemonių maršrute ar eilėje. Todėl naudojant „Python“ kalbos principus buvo sukurta klasė pavadinimu ***Car***, šios klasės pagrindinis metodas yra *drive(self)* ir „SimPy“ bibliotekos įvykio paleidiklis pavadinimu *self.canDrive*. Šio metodo ir įvykio pagalba ši klasė yra pakankamai paprasta ir lengvai gali būti pakeista, jeigu to reikėtų. Ši klasė neturi savo kintamųjų, nes yra kita klasė, kuri kuria naujas transporto priemones simuliacijoje pagal tuos kintamuosius, o klasės „Python“ kalbos kodas pateiktas 4 paveikslėlyje. Žemiau detaliau aprašyta ką atlieka ši klasė ir ką daro kiekvienas jos esantis metodas.

Klasę ***Car*** sudaro šie metodai:

- def __init__(self, env, driveTime, road, destination) – šis metodas yra konstruktorius, kuris gauna 4 įvestis, kad būtų galima sukurti šios klasės objektą. Pati pirma įvestis yra env, tai yra simuliacijos aplinkos objektas sukurtas naudojant „SimPy“ metodą *simpy.Environment()*, tuomet duodamas laikas, kiek užtruktų pervažiuoti per sankryžą iki pabaigos taško, maršrutas, kuriama ši transporto priemonė yra ir šios transporto priemonės sankryžos kelionės taškas. Įvestys *driveTime, road* ir *destination* yra gaunami iš transporto priemonių kūrėjo (angl. *generator*) pavadinimu *carGenerator*. Konstruktoriaus viduje sukuriamas vidinis ID, simuliacijos aplinka, simuliacijos įvykio sukūrimo laikas, nurodomas

sankryžos pervažiavimo laikas, nurodomas maršrutas, kuriama ši transporto priemonė yra, nurodomas kelionės tikslo taškas, sukuriamas įvykis, kada mašina gali važiuoti naudojant „SimPy“ bibliotekos metodą *env.event()* ir du statistikos kintamieji, kad būtų galima žinoti, kiek transporto priemonė laukė laiko kol galėjo įvažiuoti į sankryžą ir kiek laiko užtruko visa transporto priemonės kelionė pervažiuoti sankryžą.

- def drive(self) – šis metodas naudojamas, kad transporto priemonė galėtų važiuoti. Kai šis metodas iškviečiamas, transporto priemonė patikrina ar ji egzistuoja jai duoto maršruto sąraše, jeigu ji egzistuoja, tai reiškiasi ji laukia eilėje ir turi laukti, tol kol bus iškviestas įvykio paleidiklis pavadinimu *self.canDrive*, jeigu transporto priemonė neegzistuoja eilėje, jai nereikia laukti kada jai bus leista važiuoti ir gali pradėti savo procesą. Proceso metu atnaujinama vidinis statistinis kintamasis, kiek ši transporto priemonė turėjo laukti, kad galėtų išvažiuoti į sankryžą pavadinimu *self.waitingTime = env.now - self.startingTime*, tuomet iškviečiamas „SimPy“ bibliotekos laukimo metodas pavadinimu *yield env.timeout(self.driveTime),* kurio pagalba simuliuojama kiek transporto priemonė užtruko pervažiuoti sankryžą. Tada apskaičiuojama kiek užtruko visa transporto priemonės kelionė sankryžoje atnaujinant vidinį statistikos kintamąjį *self.journeyTime = env.now - self.startingTime* ir galiausiai iškviečiamas vidinis metodas *self.updateStats(),* kad šie vidiniai statistikos kintamieji būtų surinkti.
- def updateStats(self) – šis metodas yra skirtas surinkti statistinius duomenis: *self.waitingTime* ir *self.journeyTime*. Šie abu kintamieji yra perduodami visų transporto priemonių statistikos objektui ir būtent šio transporto priemonių maršruto statistikai surinkti.

```
class Car:
    idCounter = 0

    def __init__(self, env, driveTime, road, destination):
        self.id = "Car_" + str(Car.idCounter)
        Car.idCounter += 1

        self.env = env
        self.startingTime = env.now
        self.driveTime = driveTime
        self.road = road
        self.destination = destination
        self.canDrive = env.event()
        self.waitingTime = 0.0
```

```
        self.journeyTime = 0.0

    def drive(self):
        print("Masina %s atvyko i Masinu Kelia %i tokiu laiku %.3f" %
(self.id, self.road.start, self.startingTime))

        if self in self.road.cars:
            yield self.canDrive

        # Suskaičiuoti kiek transporto priemonė laukė (važiavimo laikas
neįskaičiuotas)
        self.waitingTime = env.now - self.startingTime

        # Simuliuojamas važiavimo laikas
        yield env.timeout(self.driveTime)

        # Suskaičiuoti koks transporto priemonės kelionės laikas (važiavimo
laikas įskaičiuotas)
        self.journeyTime = env.now - self.startingTime

        # Atnaujinti statistiką
        self.updateStats()

        print("Masina %s pasieke taska %i is Masinu Kelio %i su tokiu laiku
%.3f" % (self.id, self.destination, self.road.start, env.now))
        print("Masina %s uztruko tiek laiko %.3f" % (self.id, env.now -
self.startingTime))

    def updateStats(self):
        SIM_CAR_STATS.updateWaitingTime(self.waitingTime)
        SIM_CAR_STATS.updateJourneyTime(self.journeyTime)

        monitor.updateStats(self, self.waitingTime, self.journeyTime)
```

4 pav. Klasės Car kodas

2.2.5. Simuliacijos klasė Ped

Sankryžos simuliacijos aplinkos prototipui kitas iš svarbiausių elementų būtų pėsčiasis, kad būtų galima sekti skirtingus duomenis apie šį elementą, kartu reikia kaupti papildoma informaciją kaip: kiek užtruko laiko kelionė ar kiek užtruko laiko laukti prie raudonos šviesos. Todėl naudojant „Python“ kalbos principus buvo sukurta klasė pavadinimu ***Ped***, šios klasės pagrindinis metodas yra *walk(self)* ir „SimPy“ bibliotekos įvykio paleidiklis pavadinimu *self.canWalk*. Šio metodo ir įvykio pagalba ši klasė yra pakankamai paprasta ir lengvai gali būti pakeista, jeigu to reikėtų. Ši klasė neturi savo kintamųjų, nes yra kita klasė, kuri kuria naujus pėsčiuosius simuliacijoje pagal tuos kintamuosius, o klasės „Python“ kalbos kodas pateiktas 5 paveikslėlyje. Žemiau detaliau aprašyta ką atlieka ši klasė ir ką daro kiekvienas jos esantis metodas.

Klasę ***Ped*** sudaro šie metodai:

- def __init__(self, env, walkTime, path, destination) – šis metodas yra konstruktorius, kuris gauna 4 įvestis, kad būtų galima sukurti šios klasės objektą. Pati prima įvestis yra *env*, tai yra simuliacijos aplinkos objektas sukurtas naudojant „SimPy“ metodą *simpy.Environment()*, tuomet duodamas laikas, kiek užtruktų pereiti per pėsčiųjų perėją, pėsčiųjų perėja, kur šis pėsčiasis yra ir šios pėsčiųjų perėjos pabaigos taškas. Įvestys *walkTime*, *path* ir *destination* yra gaunami iš pėsčiųjų kūrėjo (angl. generator) pavadinimu *pedGenerator*. Konstruktoriaus viduje sukuriamas vidinis ID, simuliacijos aplinka, simuliacijos įvykio sukūrimo laikas, nurodomas pėsčiųjų perėjos perėjimo laikas, nurodoma pėsčiųjų perėja, kur šis pėsčiasis yra, nurodomas kelionės pabaigos taškas, sukuriamas įvykis, kada pėsčiasis gali eiti naudojant „SimPy“ bibliotekos metodą *env.event()* ir du statistikos kintamieji, kad būtų galima žinoti, kiek pėsčiasis laukė laiko kol galėjo eiti per pėsčiųjų perėją ir kiek laiko užtruko pėsčiajam visa kelionė pereiti per pėsčiųjų perėją.
- def walk(self) – šis metodas naudojamas, kad pėsčiasis galėtų eiti. Kai šis metodas iškviečiamas, pėsčiasis patikrina ar jis egzistuoja jai duotame pėsčiųjų perėjos sąraše. Jeigu jis egzistuoja, tai reiškiasi jis laukia eilėje ir turi laukti, kol bus iškviestas įvykio paleidiklis pavadinimu *self.canWalk*, jeigu pėsčiasis neegzistuoja eilėje, jam nereikia laukti kada jam bus leista eiti ir gali pradėti savo procesą. Proceso metu atnaujinama vidinis statistinis kintamasis pavadinimu *self.waitingTime = env.now - self.startingTime*, tuomet iškviečiamas „SimPy“ bibliotekos laukimo metodas pavadinimu *yield env.timeout(self.walkTime)*, kurio pagalba simuliuojama kiek pėsčiasis užtruko pereiti per pėsčiųjų perėją. Tada apskaičiuojama kiek užtruko visa pėsčiojo kelionė atnaujinant vidinį statistikos kintamąjį *self.journeyTime = env.now - self.startingTime* ir galiausiai iškviečiamas vidinis metodas *self.updateStats()*, kad šie vidiniai statistikos kintamieji būtų surinkti.
- def updateStats(self) – šis metodas yra skirtas surinkti statistinius duomenis: *self.waitingTime* ir *self.journeyTime*. Šie abu kintamieji yra perduodami visų pėsčiųjų statistikos objektui ir būtent šios pėsčiųjų perėjos statistikai surinkti.

```
class Ped:
    idCounter = 0
```

```
    def __init__(self, env, walkTime, path, destination):
        self.id = "Ped_" + str(Ped.idCounter)
        Ped.idCounter += 1

        self.env = env
        self.startingTime = env.now
        self.walkTime = walkTime
        self.path = path
        self.destination = destination
        self.canWalk = env.event()
        self.waitingTime = 0.0
        self.journeyTime = 0.0

    def walk(self):
        print("Pestysis %s atvyko i Pesciuju taska %i su tokiu laiku %.3f" %
(self.id, self.path.start, self.startingTime))

        if self in self.path.peds:
            yield self.canWalk

        # Suskaičiuoti kiek pėstysis laukė (ėjimo laikas neįskaičiuotas)
        self.waitingTime = env.now - self.startingTime

        # Simuliuojamas ėjimo laikas
        yield env.timeout(self.walkTime)

        # Suskaičiuoti koks pėsčiojo kelionės laikas (ėjimo laikas
įskaičiuotas)
        self.journeyTime = env.now - self.startingTime

        # Atnaujinti statistika
        self.updateStats()

        print("Pestysis %s atvyko i Pesciuju taska %i is Pesciuju tasko %i
su tokiu laiku %.3f" % (self.id, self.destination, self.path.start,
env.now))
        print("Pestysis %s uztruko tiek laiko %.3f" % (self.id, env.now -
self.startingTime))

    def updateStats(self):
        SIM_PED_STATS.updateWaitingTime(self.waitingTime)
        SIM_PED_STATS.updateJourneyTime(self.journeyTime)

        monitor.updateStats(self, self.waitingTime, self.journeyTime)
```

5 pav. Klasės Ped kodas

2.2.6. Simuliacijos klasė TrafficLight

Simuliacijos aplinkos prototipui sukurti svarbus elementas yra šviesoforas, kurio pagalba galima valdyti kada gali važiuoti transporto priemonės ir eiti pėstieji. Tuo pačiu galima valdyti transporto priemonių maršrutų ir pėsčiųjų perėjų veiksmus, kuriems leidžiama keliauti, o kuriems ne. Todėl šiam prototipui buvo sukurta šviesoforo klasė pavadinimu **TrafficLight**, kuri simuliuoja

kaip turėtų veikti tikras šviesoforas. Šios klasės pagrindinis funkcionalumas yra priskirti transporto priemonių maršrutų ir pėsčiųjų perėjų grupes, kad maršrutas arba perėja žinotų, kokia šviesa jiems šviečia ir galėtų iškviesti jų įvykių paleidimus. Paskutinis funkcionalumas yra keisti savo šviesos būseną visiems šviesoforui priklausantiems maršrutams ir perėjoms. Todėl, kad simuliacija yra diskretinių įvykių, geltonos šviesos nereikia ir ji neįtakoja kaip veikia simuliacija. Šios klasės kintamieji yra pateikti 9 lentelėje ir klasės „Python“ kalbos kodas pateiktas 6 paveikslėlyje. Žemiau detaliau aprašyta ką atlieka ši klasė ir ką daro kiekvienas jos esantis metodas.

Klasę ***TrafficLight*** sudaro šie metodai:

- def __init__(self, env, cycle, roads, paths, LinkGroups, PedLinkGroups, TIME_GREEN, TIME_RED) – šis metodas yra konstruktorius, kuris gauna 8 įvestis, kad būtų galima sukurti šios klasės objektą. Pati pirma įvestis yra env, tai yra simuliacijos aplinkos objektas sukurtas naudojant „SimPy“ metodą *simpy.Environment()*, tuomet duodamas šviesų masyvas, duodama šiam šviesoforui priklausantys transporto priemonių maršrutai, duodami šiam šviesoforui priklausančios pėsčiųjų perėjos, duodamas transporto priemonių maršrutų grupių masyvas, duodamas pėsčiųjų perėjų grupių masyvas, duodama kiek laiko užtrunka žalia šviesa ir galiausiai duodama kiek užtrunka raudona šviesa. Konstruktoriaus viduje priskiriama simuliacijos aplinka, nurodomas šviesoforo šviesų masyvas, priskiriama duoti transporto priemonių maršrutai, priskiriamos pėsčiųjų perėjos, priskiriamas transporto priemonių maršrutų grupių masyvas, priskiriamas pėsčiųjų perėjų grupių masyvas, priskiriama kiek turi šviesti žalia šviesa, priskiriama kiek turi šviesti raudona šviesa, paleidžiamas metodas *self.setGroups()*, kad būtų priskirti grupės transporto priemonių maršrutams ir pėsčiųjų perėjoms, galiausiai sukuriamas simuliacijos aplinkos procesas ir iškviečiamas metodas, kad šis šviesoforas veiktų simuliacijos aplinkoje naudojant *self.process = env.process(self.light()).*
- def setGroups(self) – šis metodas naudojamas paskirti grupes transporto priemonių maršrutams ir pėsčiųjų perėjoms. Iškviečiamas metodas *setGroupsToRoads(self)*, kad būtų sukurtos transporto priemonių maršrutų grupės, tuomet patikrinama ar yra duota bent 1 pėsčiųjų perėja, jeigu yra duota, tai iškviečiamas metodas *setGroupsToPaths(self)*, kad būtų sukurtos pėsčiųjų perėjos grupės.

- def setGroupsToRoads(self) – šis metodas priskiria šio šviesoforo grupes visiems transporto priemonių maršrutams naudojant ciklą, kuris tikrina kiekvieną maršrutą ir priskiria grupės numerį pagal kintamąjį *self.roadGroups,* kuris buvo sukurtas naudojant globalų kintamąjį *LinkGroups*.
- def setGroupsToPaths(self) – šis metodas priskiria šio šviesoforo grupes visoms pėsčiųjų perėjoms naudojant ciklą, kuris tikrina kiekvieną perėją ir priskiria grupės numerį pagal kintamąjį *self.pathGroups*, kuris buvo sukurtas naudojant globalų kintamąjį *PedLinkGroups*.
- def light(self) – šis metodas simuliuoja, kaip turėtų veikti šviesoforas, simuliacijos metu kai šis metodas pakeičia šviesoforo šviesą, visi šviesoforo transporto priemonių maršrutai ir pėsčiųjų perėjos iškviečia savo metodus, kad šviesa pasikeitė. Šio metodo viduje vyksta begalis ciklas, kurio metu patikrinama kokia pirma šviesa, jeigu tai yra žalia šviesa, tai tikrinami visi maršrutai ir keliai kurie yra 0 grupės bei jiems iškviečiami žalios šviesos metodai (transporto priemonių maršrutams – *road.greenLight()*, o pėsčiųjų perėjoms *path.greenLight()*), o visiems kitiems 1 grupės maršrutams ir perėjoms iškviečiama raudonos šviesos metodai (transporto priemonių maršrutams – *road.redLight()*, o pėsčiųjų perėjoms *path.redLight()*), tuomet šis procesas laukia, kol praeis tiek laiko, kiek turi šviesti žalia šviesa naudojant *yield env.timeout(self.greenLightTime)* metodą. Tačiau jeigu pirma šviesa yra raudona, tai tikrinamai visi maršrutai ir perėjos, kurie yra 1 grupės ir jiems iškviečiami žalios šviesos metodai, o visiems kitiems 0 grupės maršrutams ir perėjoms iškviečiami raudonos šviesos metodai, tuomet šis procesas laukia, kol praeis tiek laiko, kiek turi šviesti raudona šviesa naudojant *yield env.timeout(self.redLightTime)* metodą. Ciklo gale iškviečiamas metodas *self.switch_lights(),* kuris sumaino žalią su raudoną šviesa (Pavyzdys: [„green“, „red“] -> [„red“, „green“])
- def switch_lights(self) – Šis metodas naudojamas pakeisti savo šviesos ciklą, taip pakeičiama iš raudonos šviesos į žalią, bei atvirkščiai.

9 lentelė. Klasės TrafficLight sukurti kintamieji

Kintamieji	**Paaiškinimas**
LinkGroups = [0, 1, 0, 1]	Sąrašas, kuriame laikoma transporto priemonių maršrutų grupės. Grupes sudaro 0 arba 1, viena maršrutų grupė pradės su žalia šviesa, kita su raudona. Šis sąrašas sudarytas, pagal tą patį formatą kaip nukreipto grafo kraštų sąrašas, todėl reikia atsižvelgti į *Links* kintamąjį. Šiuo atveju grupės būtų tokios: 0 maršrutas yra 0 grupės, 2 maršrutas yra 1 grupės, 4 maršrutas yra 0 grupės ir 6 maršrutas yra 1 grupės.
PedLinkGroups = [1, 1, 0, 0, 1, 1, 0, 0]	Sąrašas, kuriame laikoma pėsčiųjų perėjų grupės. Grupes sudaro 0 arba 1, viena perėjų grupė pradės su žalia šviesa, kita su raudona. Šis sąrašas sudarytas, pagal tą patį formatą kaip nukreipto grafo kraštų sąrašas, todėl reikia atsižvelgti į *PedLinks* kintamąjį. Šiuo atveju grupės būtų tokios: [0, 1] ir [1, 0] yra 1 grupės, [1, 2] ir [2, 1] yra 0 grupės, [2, 3] ir [3, 2] yra 1 grupės, [3, 0] ir [0, 3] yra 0 grupės.
TIME_GREEN = 35	Kintamasis, kuriame nurodama kiek laiko (sekundėmis) šviečia žalia šviesa.
TIME_RED = 20	Kintamasis, kuriame nurodama kiek laiko (sekundėmis) šviečia raudona šviesa.

```
class TrafficLight:
    def __init__(self, env, cycle, roads, paths, LinkGroups, PedLinkGroups,
TIME_GREEN, TIME_RED):
        self.env = env
        self.cycle = cycle
        self.roads = roads
        self.paths = paths
        self.roadGroups = LinkGroups
        self.pathGroups = PedLinkGroups
        self.greenLightTime = TIME_GREEN
```

```
        self.redLightTime = TIME_RED

        self.setGroups()

        self.process = env.process(self.light())

    def setGroups(self):
        self.setGroupsToRoads()
        if len(self.paths) > 0:
            self.setGroupsToPaths()

    def setGroupsToRoads(self):
        i = 0
        for road in self.roads:
            road.setGroup(self.roadGroups[i])
            i = i + 1

    def setGroupsToPaths(self):
        i = 0
        for path in self.paths:
            path.setGroup(self.pathGroups[i])
            i = i + 1

    def light(self):
        while True:
            # Dabartinė šviesoforo šviesa
            print("\nŠviesoforo pasikeitė šviesa %s tokiu laiku %.3f." %
(self.cycle[0], env.now))
            # Jeigu žalia, tai visiems 0 grupės keliams leisti važiuoti
            if self.cycle[0] == 'green':
                # Kelių triggeriai
                for road in self.roads:
                    if road.group == 0:
                        road.greenLight()
                    else:
                        road.redLight()
                # Pėsčiųjų kelių triggeriai
                for path in self.paths:
                    if path.group == 0:
                        path.greenLight()
                    else:
                        path.redLight()

                yield env.timeout(self.greenLightTime)
            # Jeigu raudona, tai visiems 1 grupės keliams leisti važiuoti
            else:
                # Kelių triggeriai
                for road in self.roads:
                    if road.group == 1:
                        road.greenLight()
                    else:
                        road.redLight()
                # Pėsčiųjų kelių triggeriai
                for path in self.paths:
                    if path.group == 1:
                        path.greenLight()
                    else:
                        path.redLight()
```

```
            yield env.timeout(self.redLightTime)
        self.switch_lights()

def switch_lights(self):
    self.cycle[0], self.cycle[1] = self.cycle[1], self.cycle[0]
```

6 pav. Klasės TrafficLight kodas

2.2.7. Simuliacijos klasė CarGenerator

Simuliacijos aplinka turi jau visas reikalingas klases, kad galėtų būti sukurtas sankryžos simuliacijos aplinkos prototipas, tačiau reikia papildomos klasės, kuri galėtų sukurti naujas transporto priemones, kas tam tikrą laiką. Todėl buvo sukurta papildoma klasė pavadinimu **CarGenerator**, kad nereikėtų kurti transporto priemonių rankiniu būdu ir kad sukurtos transporto priemonės turėtų tam tikrus rodiklius, kurių pagalba galima valdyti kur jos važiuoja. Ši klasė turi daugelį savo kintamųjų, kurių pagalba galima nustatyti, kurie transporto priemonių maršrutai turi didesnį transporto priemonių srautą, iš kurio taško į kurį tašką dažniausiai važiuoja transporto priemonės (pvz.: tiesiai, į dešinę arba į kairę), kiek kiekvienas posūkis užtrunka. Šios klasės pagrindinis veikimas yra sukurti naujas transporto priemonės ant pateiktų maršrutų su tam tikra tikimybę, transporto priemonė gali pasirinkti kur nori važiuoti iš sukurto maršruto pradžios taško su tam tikra tikimybę ir galiausiai naudojant Puasono procesą sukurti transporto priemones, kas tam tikrą intervalą. Šios klasės kintamieji yra pateikti 10 lentelėje ir klasės „Python“ kalbos kodas pateiktas 7 paveikslėlyje. Žemiau detaliau aprašyta ką atlieka ši klasė ir ką daro kiekvienas jos esantis metodas.

Klasę ***CarGenerator*** sudaro šie metodai:

- def __init__(self, env, roads, RoadWeights, DestinationWeights, TurnTimes, ARRIVAL_MEAN) – šis metodas yra konstruktorius, kuris gauna 6 įvestis, kad būtų galima sukurti šios klasės objektą. Pati pirma įvestis yra env, tai yra simuliacijos aplinkos objektas sukurtas naudojant „SimPy“ metodą *simpy.Environment()*, tuomet duodami transporto priemonių maršrutai, kuriuose gali sukurti transporto priemones, duodami transporto priemonių maršrutų svoriai, kad būtų galima nurodyti, kokie maršrutai gauna daugiau transporto priemonių, duodami kelionės pabaigos taškų svoriai, kad būtų žinoma į kokius posūkius dažniau suka, duodami laikai, kiek užtrunka atlikti posūkius, galiausiai duodamas laikas pagal kurį nustatoma, kiek laiko turi užtruki tarp dviejų mašinų atsiradimo.

Konstruktoriaus viduje priskiriama simuliacijos aplinka, nurodomi transporto priemonių maršrutai, nurodomos maršrutų tikimybės, nurodomos maršrutų pabaigos taškų tikimybės, nurodomi maršrutų posūkių užtrukimo laikai, nurodoma kas kiek laiko turi būti sukurta nauja transporto priemonė, skaičiuojama kiek transporto priemonių buvo sukurta ir galiausiai sukuriamas naujas simuliacijos aplinkos procesas ir iškviečiamas transporto priemonių kūrimo metodas *self.process = env.process(self.generate())*.

- def generate(self) – šis metodas yra pats pagrindinis, jo viduje vyksta begalinis ciklas, kurio metu pasirenkamas atsitiktinis maršrutas pagal pateiktas tikimybes *WrandomRoad = random.choice(len(self.roads), 1, p=self.roadWeights)*. Kai yra žinomas pasirinktas maršrutas tuomet atsitiktinai pasirenkamas transporto priemonės maršruto pabaigos taškas pagal pateiktas tikimybes *WrandomDest = random.choice(len(self.roads[WrandomRoad[0]].ends), 1, p=self.destinationWeights)*. Taip gaunama kelionė, kuri bus perduodama sukurtai transporto priemonei, tuo pačiu metu apskaičiuojama kiek laiko užtruktų atlikti posūkį sankryžoje naudojant pateiktus laikus *turnTime = self.turnTimes[WrandomDest[0]]*. Tuomet sukuriamas naujas transporto priemonės objektas naudojant *WrandomRoad, WrandomDest* ir *turnTime* kintamuosius ir ši transporto priemonė yra pridedama į pasirinktą maršrutą naudojant kelio *addCar()* metodą. Galiausiai atnaujinami globalūs statistikos duomenys ir laukiama pagal Puasono procesą tam tikrą laiką naudojant „SimPy“ laukimo metodą ir „NumPy“ Puasono pasiskirstymo metodą *yield env.timeout(random.exponential(self.arrivalTime))*.

10 lentelė. Klasės CarGenerator sukurti kintamieji

Kintamieji	**Paaiškinimas**
RoadWeights = [0.35, 0.15, 0.35, 0.15]	Sąrašas, kuriame laikomos tikimybės, kurios nurodo kokie maršrutai turi kokia tikimybę, kad ten atvažiuos transporto priemonė. Formatas sudarytas pagal *Links* kintamąjį, todėl formatas būtų toks:

Kintamieji	Paaiškinimas
	maršrutas 0 turi 35% tikimybę, maršrutas 2 turi 15% tikimybę, maršrutas 4 turi 35% tikimybę ir maršrutas 6 turi 15% tikimybę, kad atvažiuos transporto priemonė.
DestinationWeights = [0.40, 0.40, 0.20]	Sąrašas, kuriame laikomos tikimybės, kurios nurodo kokie posūkiai turi kokia tikimybę, kad ten pasuks transporto priemonė. Formatas sudarytas pagal Links kintamąjį ir galimus pabaigos taškus, todėl formatas būtų toks: posūkis [0, 5] turi 40% tikimybę, posūkis [0, 7] turi 40% tikimybę, posūkis turi [0, 3] turi 20% tikimybę ir t.t.
TurnTimes = [[T_DEPART_MIDDLE, T_DEPART_RIGHT, T_DEPART_LEFT]]	Sąrašas, kuriama laikomas formatas ir laikai, kiek užtrunka kiekvienas transporto priemonės posūkis. Jeigu yra maršrutų, kurie turi mažiau posūkių, tai tuomet galima pridėti naują sąrašą viduje ir tik jo dalį perduoti *CarGenerator*.
T_DEPART_RIGHT = 1.6 T_DEPART_MIDDLE = 2.0 T_DEPART_LEFT = 2.4	Kintamieji kurie parodo kiek kiekvienas posūkis užtrunka laiko.
ARRIVAL_MEAN = 5	Kintamasis kuris parodo, kas kiek sekundžių vidutiniškai turėtų būti sukurta nauja transporto priemonė. Klasė *CarGenerator* naudoja Puasono pasiskirstymą.

```
class CarGenerator:
    def __init__(self, env, roads, RoadWeights, DestinationWeights,
TurnTimes, ARRIVAL_MEAN):
        self.env = env
        self.roads = roads
        self.roadWeights = RoadWeights
```

```
        self.destinationWeights = DestinationWeights
        self.turnTimes = TurnTimes
        self.arrivalTime = ARRIVAL_MEAN
        self.carCount = 0

        self.process = env.process(self.generate())

    def generate(self):
        # Sukurk naujas mašinas iki simuliacijos galo
        while True:
            # Pagal svorius pasirenkamas kelias
            WrandomRoad = np.random.choice(len(self.roads), 1,
p=self.roadWeights)

            if len(self.roads[WrandomRoad[0]].ends) == 1:
                WrandomDest =
np.random.choice(len(self.roads[WrandomRoad[0]].ends), 1,
p=self.destinationWeights[1])
                turnTime = self.turnTimes[1][WrandomDest[0]]
            else:
                # Pagal svorius parenkamas pasirinkto kelio kelionės taškas
                WrandomDest =
np.random.choice(len(self.roads[WrandomRoad[0]].ends), 1,
p=self.destinationWeights[0])
                # Patikrinama kiek laiko turi užtrukti posūkis
                turnTime = self.turnTimes[0][WrandomDest[0]]

            # Sukuriama nauja mašina su duomenimis: WrandomRoad, WrandomDest
ir turnTime
            c = Car(env, turnTime, self.roads[WrandomRoad[0]],
self.roads[WrandomRoad[0]].ends[WrandomDest[0]])
            self.roads[WrandomRoad[0]].addCar(c)

            # Skaičiuojama kiek buvo sukurtu mašinu
            self.carCount += 1
            SIM_CAR_STATS.updateCount(1)

            # Pagal Poisson procesą laukiama kol atvyks nauja mašina
            yield env.timeout(np.random.exponential(self.arrivalTime))
```

7 pav. Klasės CarGenerator kodas

2.2.8. Simuliacijos klasė PedGenerator

Simuliacijos klasė pavadinimu **PedGenerator** atlieka tokį patį funkcionalumą kaip **CarGenerator** klasė, tik tiek, kad ji kuria pagal Puasono procesą pėsčiuosius vietoj transporto priemonių ir reikalauja mažiau įvesties kintamųjų, kad galėtų tai atlikti, nes pėsčiųjų perėjos yra paprastesnės negu transporto priemonių maršrutai. Šios klasės kintamieji yra pateikti 11 lentelėje ir klasės „Python“ kalbos kodas pateiktas 8 paveikslėlyje. Žemiau detaliau aprašyta ką atlieka ši klasė ir ką daro kiekvienas jos esantis metodas.

Klasę ***PedGenerator*** sudaro šie metodai:

- def __init__(self, env, paths, PathWeights, T_PED_DEPART, PED_ARRIVAL_MEAN) – šis metodas yra konstruktorius, kuris gauna 5 įvestis, kad būtų galima sukruti šios klasės objektą. Pati pirma įvestis yra env, tai yra simuliacijos aplinkos objektas sukurtas naudojant „SimPy“ metodą *simpy.Environment()*, tuomet duodamos pėsčiųjų perėjos, kuriose gali sukurti pėsčiuosius, duodami pėsčiųjų perėjų svoriai, kad būtų galima nurodyti, kokios perėjos gauna daugiau pėsčiųjų. Duodama kiek laiko užtrunka pereiti per pėsčiųjų perėją ir duodama kas kiek laiko vidutiniškai turi būti sukurtas naujas pėsčiasis. Konstruktoriaus viduje priskiriama simuliacijos aplinka, nurodomos pėsčiųjų perėjos, pėsčiųjų perėjų tikimybės, nurodoma kiek užtrunka laiko pereiti per perėją, nurodoma kas kiek laiko turi būti sukurtas naujas pėsčiasis, skaičiuojama kiek pėsčiųjų buvo sukurta ir galiausiai sukuriamas naujas simuliacijos aplinkos procesas ir iškviečiamas pėsčiųjų kūrimo metodas *self.process = env.process(self.generate())*.
- def generate(self) – šis metodas yra pats pagrindinis šios klasės metodas, šio metodo viduje vyksta be galinis ciklas, kurio metu pasirenkamas atsitiktinė pėsčiųjų perėja naudojant pateiktas pėsčiųjų perėjų tikimybes WrandomPath = random.choice(len(self.paths), 1, p=self.pathWeights). Dėl to, kad pėsčiųjų perėjos yra daug paprastesnės su vienu kintamuoju, pėsčiųjų perėjos pradžia ir pabaiga, todėl galima sukurti pėsčiojo objektą ir jį pridėti į pėsčiųjų perėjos kelią naudojant pėsčiųjų perėjos metodą addPed(). Galiausiai atnaujinami globalūs statistikos duomenys ir laukiama pagal Puasono procesą tam tikrą laiką naudojant „SimPy“ laukimo metodą ir „NumPy“ Puasono pasiskirstymo metodą *yield env.timeout(random.exponential(self.arrivalTime))*.

11 lentelė. Klasės PedGenerator sukurti kintamieji

Kintamieji	**Paaiškinimas**
PathWeights = [0.125, 0.125, 0.125, 0.125, 0.125, 0.125, 0.125, 0.125]	Sąrašas, kuriame laikomos tikimybės, kurios nurodo kokios perėjos turi kokia tikimybę, kad ten ateis pėsčiasis. Formatas sudarytas pagal *PedLinks*

Kintamieji	Paaiškinimas
	kintamąjį, todėl formatas būtų toks: perėja [0, 1] turi 12.5% tikimybę, perėja [1, 0] turi 12.5% tikimybę, perėja [1, 2] turi 12.5% tikimybę ir perėja [2, 1] turi 12.5% tikimybę ir t.t., kad ateis pėsčiasis.
T_PED_DEPART = 3.0	Kintamasis kuris užtrunka pereiti per perėja.
PED_ARRIVAL_MEAN = 10	Kintamasis kuris parodo, kas kiek sekundžių vidutiniškai turėtų būti sukurtas naujas pėsčiasis. Klasė Ped*Generator* naudoja Puasono pasiskirstymą.

```
class PedGenerator():
    def __init__(self, env, paths, PathWeights, T_PED_DEPART,
PED_ARRIVAL_MEAN):
        self.env = env
        self.paths = paths
        self.pathWeights = PathWeights
        self.arrivalTime = PED_ARRIVAL_MEAN
        self.departTime = T_PED_DEPART
        self.pedCount = 0

        self.process = env.process(self.generate())

    def generate(self):
        # Sukurk naujus pėsčiuosius iki simuliacijos galo
        while True:
            # Pagal svorius pasirenkamas kelias
            WrandomPath = np.random.choice(len(self.paths), 1,
p=self.pathWeights)

            # Sukuriamas naujas pėstysis su duomenimis: WrandomPath
            p = Ped(env, self.departTime, self.paths[WrandomPath[0]],
self.paths[WrandomPath[0]].end)
            self.paths[WrandomPath[0]].addPed(p)

            # Skaičiuojama kiek buvo sukurta pėsčiųjų
            self.pedCount += 1
            SIM_PED_STATS.updateCount(1)

            # Pagal Poisson procesą laukiama kol atvyks naujas pėstysis
            yield env.timeout(np.random.exponential(self.arrivalTime))
```

8 pav. Klasės PedGenerator kodas

2.2.9. Klasės Stats ir Monitor

Šios dvi klasės yra paskutinės šiam darbui atlikti. Klasės pavadinimu **Stats** tikslas yra surinkti bendrus statistinius duomenis kaip: kiek buvo sukurta objektų, kiek kiekvienas objektas laukė, kiek kiekvienas objektas užtruko atlikti savo kelionę, objekto laukimo laiko vidurkis ir galiausiai objekto kelionės užtrūkimo vidurkis. Todėl galima sakyti, kad klasė **Stats** tiesiog yra duomenų struktūra, kuri gali būti bendra visai simuliacijai, arba kiekvienam transporto priemonės maršrutui arba pėsčiųjų perėjai. Būtent todėl klasė **Monitor** naudoją klasę **Stats**, kad galėtų surinkti statistinius duomenis apie kiekviena simuliacijos aplinkoje esantį transporto priemonės maršrutą ir pėsčiųjų perėją. Galiausiai simuliacijos pabaigoje ši klasė sukuria diagramas naudojant „matplotlib“ biblioteką, pateikiant kiekvieno maršruto ir perėjos laukimo diagramą, jų laukimo vidurkį, kad būtų galima matyti simuliacijos aplinkos pakeitimų įtaką. Klasės **Monitor** „Pyhton“ kalbos kodas pateiktas 9 paveikslėlyje, o žemiau detaliau aprašyta ką atlieka ši klasė ir ką daro kiekvienas jos esantis metodas.

Klasę ***Monitor*** sudaro šie metodai:

- def __init__(self, roads, paths) – šis metodas yra konstruktorius, kad būtų galima sukurti šį objektą, reikia pateikti 2 įvesties elementus. Pirmas įvesties elementas yra transporto priemonių maršrutai, kitas įvesties elementas yra pėsčiųjų perėjos, turint šiuos elementus galima sukurti jų statistikas šioje klasėje. Konstruktoriaus viduje priskiriami transporto priemonių maršrutai, priskiriamos pėsčiųjų perėjos, sukuriamas statistikos sąrašas, sukuriamas ID sąrašas ir galiausiai iškviečiamas metodas *self.makeMonitors()*, kad būtų sukurta statistikos klasė **Stats** kiekvienam iš kelių.
- def makeMonitors(self) – šis metodas patikrina kiek duotą maršrutų ir perėjų, jeigu daugiau negu vienas tai gali leisti sukurti statistikos objektus kiekvienam iš jų. Iškviečiami abu metodai *self.roadMonitors()* ir *self.pathMonitors()*, jeigu nėra daugiau negu vieno maršruto arba perėjos šie metodai nėra iškviečiami, nes nėra ko rinkti.
- def roadMonitors(self) – šio metodo viduje pereinama per visus pateiktus transporto priemonių maršrutus ir sukuriama kiekvienam iš jų **Stats** klasės objektas, kuriama bus kaupiama statistika apie šį maršrutą.

- def pathMonitors(self) - šio metodo viduje pereinama per visas pateiktas pėsčiųjų perėjas ir sukuriama kiekvienam iš jų **Stats** klasės objektas, kuriama bus kaupiama statistika apie šią perėją.
- def updateStats(self, obj) – šio metodo pagalba surenkama statistiniai duomenys apie kiekvieną maršrutą ir perėją. Šiam metodui yra duodamas objektas, kuris gali būti transporto priemonė arba pėsčiasis. Patikrinama ar paduotas objektas yra transporto priemonė, jeigu taip paimama šio objekto surinkta statistika *obj.waitingTime* ir *obj.journeyTime* ir perduodama statistikos kaupimo objektui, kuris renka būtent šios transporto priemonės keliavimo maršruto statistiką. Tuomet jeigu objektas yra pėsčiasis tai surenkama tokia pati statistika, tik tiek, kad ji yra perduodama kaupimo objektui, kuris renka būtent to pėsčiųjų perėjos statistiką.
- def createPlot(self) – pagrindinis šios klasės metodas, šio metodo viduje yra sukuriama du skirtingi diagramų plotai, viename jų atvaizduojami visų transporto priemonių maršrutų diagramos, o kitame visų pėsčiųjų perėjų diagramos. Pirmosios diagramos atvaizduoja kiek laukia kiekvienas objektas savo maršrute arba perėjoje, kartu kiekvienoje diagramos dalyje pažymimas vidutinis laukimo laikas. Antros diagramos atvaizduoja kiek laukia kiekvienas objektas savo maršrute arba perėjoje histogramos formatu, kad būtų galima pamatyti įvykių kiekį kiekviename laiko tarpe, tuo pačiu kiekvienoje diagramos dalyje pažymimas vidutinis laukimo laikas. Galiausiai visos diagramos yra sujungiamos poromis, kad būtų patogiau analizuoti diagramas, pavyzdžiui: pėsčiųjų perėjos yra sujungiamos, jeigu jos eina per tą pačia pėsčiųjų perėja, bet iš skirtingų kelio pusių, tuo tarpu transporto priemonių maršrutai yra sujungiami, priešingai jiems esančiais maršrutais, nes jie sudaro tą patį kelią.

```
class Monitor:
    def __init__(self, roads, paths):
        self.roads = roads
        self.paths = paths
        self.stats = []
        self.idList = []

        self.makeMonitors()

    def makeMonitors(self):
        if len(self.roads) > 0:
            self.roadMonitors()
        if len(self.paths) > 0:
```

```
            self.pathMonitors()

    def roadMonitors(self):
        for road in self.roads:
            s = Stats()
            self.stats.append(s)
            self.idList.append(road.id)

    def pathMonitors(self):
        for path in self.paths:
            s = Stats()
            self.stats.append(s)
            self.idList.append(path.id)

    def updateStats(self, obj):
        if isinstance(obj, Car):
            if obj.road in self.roads:
                statsIndex = self.idList.index(obj.road.id)
                self.stats[statsIndex].updateCount(1)
                self.stats[statsIndex].updateWaitingTime(obj.waitingTime)

                self.stats[statsIndex].updateJourneyTime(obj.journeyTime)
        elif isinstance(obj, Ped):
            if obj.path in self.paths:
                statsIndex = self.idList.index(obj.path.id)
                self.stats[statsIndex].updateCount(1)
                self.stats[statsIndex].updateWaitingTime(obj.waitingTime)

                self.stats[statsIndex].updateJourneyTime(obj.journeyTime)

    def createPlot(self):
        roadCount = int(len(self.roads) / 2)
        pathCount = int(len(self.paths) / 2)

        fig1 = plt.figure("Transporto priemonių laukimo laikas (TIME_GREEN =
%is, TIME_RED = %is, SIMULATION_TIME = %is)" % (TIME_GREEN, TIME_RED,
SIMULATION_TIME))
        for i in range(roadCount):
            stat = self.stats[i]
            stat2 = self.stats[i + 2]

            data = [i for i in stat.waitingTimes if i != 0]
            data2 = [i for i in stat2.waitingTimes if i != 0]

            x = list(range(1, (len(data) + 1)))
            x2 = list(range(1, (len(data2) + 1)))

            average = sum(data) / float(len(data))
            average2 = sum(data2) / float(len(data2))

            plt.subplot(math.floor(roadCount), math.ceil(roadCount / 2), (i
+ 1))

            plt.plot(x, data, color='b')
            plt.plot(x2, data2, color='g')

            plt.axhline(y=average, color='b', linestyle=':', label='M1
vidurkis %.2f' % average)
```

```
            plt.axhline(y=average2, color='g', linestyle=':', label='M2
vidurkis %.2f' % average2)

            plt.legend(loc='upper right')

            fillteredRoads = [road for road in self.roads if road.id ==
self.idList[i] or road.id == self.idList[i + 2]]
            plt.title("Maršrutai: M1= %i: %s ir M2= %i: %s" %
(fillteredRoads[0].start, str(fillteredRoads[0].ends),
fillteredRoads[1].start, str(fillteredRoads[1].ends)))

        fig1.suptitle("Transporto priemonių laukimo laikas (TIME_GREEN =
%is, TIME_RED = %is, SIMULATION_TIME = %is)" % (TIME_GREEN, TIME_RED,
SIMULATION_TIME), fontsize=16)
        fig1.supxlabel("Transporto priemonių kiekis")
        fig1.supylabel("Laukimo laikas sekundėmis")

        fig2 = plt.figure("Pesčiųjų perėjų laukimo laikas (TIME_GREEN = %is,
TIME_RED = %is, SIMULATION_TIME = %is)" % (TIME_GREEN, TIME_RED,
SIMULATION_TIME))
        count = roadCount * 2
        for i in range(pathCount):
            stat = self.stats[count + i]
            stat2 = self.stats[count + i + 1]

            data = [i for i in stat.waitingTimes if i != 0]
            data2 = [i for i in stat2.waitingTimes if i != 0]

            average = sum(data) / float(len(data))
            average2 = sum(data2) / float(len(data2))

            x = list(range(1, (len(data) + 1)))
            x2 = list(range(1, (len(data2) + 1)))

            plt.subplot(math.floor(pathCount / 2), math.ceil(pathCount / 2),
(i + 1))

            plt.plot(x, data, color='b')
            plt.plot(x2, data2, color='g')

            plt.axhline(y=average, color='b', linestyle=':', label='P1
vidurkis %.2f' % average)
            plt.axhline(y=average2, color='g', linestyle=':', label='P2
vidurkis %.2f' % average2)

            plt.legend(loc='upper right')

            fillteredPaths = [path for path in self.paths if path.id ==
self.idList[count + i] or path.id == self.idList[count + i + 1]]

            count = count + 1

            plt.title("Perėjos: P1= [%i, %i], P2= [%i, %i]" %
(fillteredPaths[0].start, fillteredPaths[0].end, fillteredPaths[1].start,
fillteredPaths[1].end))
```

```
        fig2.suptitle("Pesčiųjų perėjų laukimo laikas (TIME_GREEN = %is,
TIME_RED = %is, SIMULATION_TIME = %is)" % (TIME_GREEN, TIME_RED,
SIMULATION_TIME), fontsize=16)
        fig2.supxlabel("Pėsčiųjų kiekis")
        fig2.supylabel("Laukimo laikas sekundėmis")

        fig3 = plt.figure("Transporto priemonių laukimo laiko histograma
(TIME_GREEN = %is, TIME_RED = %is, SIMULATION_TIME = %is)" % (TIME_GREEN,
TIME_RED, SIMULATION_TIME))
        for i in range(roadCount):
            stat = self.stats[i]
            stat2 = self.stats[i + 2]

            data = [i for i in stat.waitingTimes if i != 0]
            data2 = [i for i in stat2.waitingTimes if i != 0]

            average = sum(data) / float(len(data))
            average2 = sum(data2) / float(len(data2))

            maxvalue = int(max(data + data2))
            bins = np.linspace(0, maxvalue, maxvalue)

            plt.subplot(math.floor(roadCount), math.ceil(roadCount / 2), (i
+ 1))

            plt.hist(data, bins, color='b', alpha=0.5)
            plt.hist(data2, bins, color='g', alpha=0.5)

            plt.axvline(x=average, color='b', linestyle=':', label='M1
vidurkis %.2f' % average)
            plt.axvline(x=average2, color='g', linestyle=':', label='M2
vidurkis %.2f' % average2)

            plt.legend(loc='upper right')

            fillteredRoads = [road for road in self.roads if road.id ==
self.idList[i] or road.id == self.idList[i + 2]]
            plt.title("Maršrutai: M1= %i: %s ir M2= %i: %s" %
(fillteredRoads[0].start, str(fillteredRoads[0].ends),
fillteredRoads[1].start, str(fillteredRoads[1].ends)))

        fig3.suptitle("Transporto priemonių laukimo laiko histograma
(TIME_GREEN = %is, TIME_RED = %is, SIMULATION_TIME = %is)" % (TIME_GREEN,
TIME_RED, SIMULATION_TIME), fontsize=16)
        fig3.supxlabel("Laukimo laikas sekundėmis")
        fig3.supylabel("Transporto priemonių kiekis")

        fig4 = plt.figure("Pesčiųjų perėjų laukimo laiko histograma
(TIME_GREEN = %is, TIME_RED = %is, SIMULATION_TIME = %is)" % (TIME_GREEN,
TIME_RED, SIMULATION_TIME))
        count = roadCount * 2
        for i in range(pathCount):
            stat = self.stats[count + i]
            stat2 = self.stats[count + i + 1]

            data = [i for i in stat.waitingTimes if i != 0]
            data2 = [i for i in stat2.waitingTimes if i != 0]
```

```
            average = sum(data) / float(len(data))
            average2 = sum(data2) / float(len(data2))

            maxvalue = int(max(data + data2))
            bins = np.linspace(0, maxvalue, maxvalue)

            plt.subplot(math.floor(pathCount / 2), math.ceil(pathCount / 2),
(i + 1))

            plt.hist(data, bins, color='b', alpha=0.5)
            plt.hist(data2, bins, color='g', alpha=0.5)

            plt.axvline(x=average, color='b', linestyle=':', label='P1
vidurkis %.2f' % average)
            plt.axvline(x=average2, color='g', linestyle=':', label='P2
vidurkis %.2f' % average2)

            plt.legend(loc='upper right')

            fillteredPaths = [path for path in self.paths if path.id ==
self.idList[count + i] or path.id == self.idList[count + i + 1]]

            count = count + 1

            plt.title("Perėjos: P1= [%i, %i], P2= [%i, %i]" %
(fillteredPaths[0].start, fillteredPaths[0].end, fillteredPaths[1].start,
fillteredPaths[1].end))

        fig4.suptitle("Pesčiųjų perėjų laukimo laiko histograma (TIME_GREEN
= %is, TIME_RED = %is, SIMULATION_TIME = %is)" % (TIME_GREEN, TIME_RED,
SIMULATION_TIME), fontsize=16)
        fig4.supxlabel("Laukimo laikas sekundėmis")
        fig4.supylabel("Pėsčiųjų kiekis")

        plt.show()

            plt.subplot(math.floor(pathCount / 4), math.ceil(pathCount / 2),
(i + 1))
            plt.hist(data, bins=math.ceil(maxvalue/1))
            plt.axvline(x=stat.calculateAverageWaitingTime(), color='r',
linestyle='-', label='Vidurkis %.2f' % stat.calculateAverageWaitingTime())
            path = [path for path in self.paths if path.id ==
self.idList[roadCount + i]]
            plt.title("Pradžia %i, pabaiga %i" % (path[0].start,
path[0].end))

        fig4.suptitle("Pesčiųjų kelių laukimo laiko histograma, kai žalia
šviečia %is, o raudona %is. Simuliacija vyksta %is" % (TIME_GREEN, TIME_RED,
SIMULATION_TIME), fontsize=16)
        fig4.legend()
        fig4.supxlabel("Laukimo laikas senkundėmis")
        fig4.supylabel("Pėsčiųjų kiekis")

        plt.show()
```

9 pav. Klasės Monitor kodas

3. Verifikavimas ir vertinimas

Šiame skyriuje bus verifikuojamas sukurtas sankryžos simuliacijos aplinkos prototipas ir kartu bus pateikti keli variantai pagerinti šį prototipą. Tuomet bus tikrinami visi simuliacijos aplinkos variantai, simuliuojant sankryžos streso testą, kas prilygtų piko valandai. Verifikavimui bus naudojamos statistinės diagramos ir surinkti statistiniai duomenys simuliacijos metu. Sankryžos prototipas sukurtas pagal dviejų kelių susikirtimo sankryža, kuri yra kontroliuojama šviesoforu ir pėstieji gali pereiti per kelią vadovaujantis šviesoforu. Šioje sankryžoje yra pagrindinis kelias, kuriame visada bus didesnis srautas transporto priemonių negu kitame. Šios sankryžos prototipo modelis yra pateiktas kaip nukreipiamasis grafas 1 paveikslėlyje, sankryžą sudaro du keliai, kurie turi tik po vieną kelio liniją kairėje ir dešinėje kelio pusėse, tuo pačiu pėstieji gali eiti per visus kelius sulaukdami šviesoforo žalios šviesos.

Simuliacijos aplinkos metu įeina daugelis skirtingų kintamųjų, kurių dėka gali pasikeisti simuliacijos rezultatai ir pati simuliacijoje esanti aplinka. Todėl 12 lentelėje yra pateikti visi kintamieji ir jų paaiškinimai, kad būtų žinoma pradinės simuliacijos aplinkos prototipo kintamieji. Taip bus galima pagerinti prototipą tik pakeičiant simuliacijos aplinkos kintamuosius. Šiame skyriuje bus aprašytas pradinis prototipas, tuomet bus pasiūlytas prototipo pagerinimas pakeičiant tik šviesoforo laikus ir galiausiai bus pasiūlytas prototipo variantas, kur yra papildomų kelių ten kur laukia ilgiausiai transporto priemonės.

12 lentelė. Simuliacijos aplinkos pradinio prototipo pagrindiniai kintamieji

Kintamasis	**Paaiškinimas**
RANDOM_SEED = 10	Simuliacijos aplinkos kintamasis, kurio pagalba galim atkurti rezultatus.
SIMULATION_TIME = 3600	Simuliacijos aplinkos simuliavimo laikas, šiuo atveju simuliacija vyksta 3600 sekundžių arba lygiai 1 valandą.
TIME_GREEN = 30	Simuliacijos aplinkos šviesoforo žalios šviesos būsenos laikas.
TIME_RED = 30	Simuliacijos aplinkos šviesoforo raudonos šviesos būsenos laikas.

Kintamasis	**Paaiškinimas**
ARRIVAL_MEAN = 5	Simuliacijos aplinkos kintamasis, kuris parodo, kas kiek laiko vidutiniškai atvyksta transporto priemonė.
PED_ARRIVAL_MEAN = 10	Simuliacijos aplinkos kintamasis, kuris parodo, kas kiek laiko vidutiniškai atvyksta pėsčiasis.
Links = {0: [5, 7, 3], 2: [7, 1, 5], 4: [1, 3, 7], 6: [3, 5, 1]}	Simuliacijos aplinkos nukreipiamasis grafas, kuriame matoma kokie yra transporto priemonių maršrutai ir kokie galimi maršruto išvažiavimai. Šis kintamasis sukurtas iš 1 pav.
PedLinks = [[0, 1], [1, 0], [1, 2], [2, 1], [2, 3], [3, 2], [3, 0], [0, 3]]	Simuliacijos aplinkos nukreipiamasis grafas, kuriame matoma kokios yra pėsčiųjų perėjos ir kokie galimi perėjos išėjimai. Šis kintamasis sukurtas iš 1 pav.
RoadWeights = [0.35, 0.15, 0.35, 0.15]	Simuliacijos aplinkos tikimybės transporto priemonėms atvažiuoti į tam tikrą transporto priemonių maršrutą.
PathWeights = [0.125, 0.125, 0.125, 0.125, 0.125, 0.125, 0.125, 0.125]	Simuliacijos aplinkos tikimybės pėstiesiems ateiti į tam tikrą pėsčiųjų perėją.

Naudojantis aukščiau pateikta lentele, galime matyti, jog tai būtų įprasta sankryžą, kurioje yra visos pėsčiųjų perėjos ir iš kiekvieno transporto priemonės kelio galima važiuoti į visas puses (tiesiai, į dešinė ir į kairę). Simuliacija vyksta 3600 sekundžių (1 valandą), šios simuliacijos atkūrimo sėkla (seed) yra 10, simuliacijos metu atvyksta naujos transporto priemonės kas 5 sekundes, naudojant Puasono pasiskirstymą t. y. transporto priemonė gali atvykti anksčiau negu 5 sekundės ir vėliau negu 5 sekundės, tačiau vidurkis išlieka maždaug 5 sekundės, taip gaunami diskretūs įvykiai, kurie nėra priklausimo vienas nuo kito. Tas pats principas veikia ir pėsčiųjų atvykimui nurodyti, tik tiek kad jie atvyksta kas 10 sekundžių. Galiausiai transporto priemonių maršrutų svoriai *RoadWeights* parodo kad maršrutas 0 turi 35%, maršrutas 2 turi 15%, maršrutas 4 turi 35% ir maršrutas 6 turi 15% visų atvykstančių transporto priemonių kiekio. Tas pats vyksta ir su pėsčiųjų perėjomis, tik šiuo atveju bet kuriame pėsčiųjų perėjos pradžios taške gali atvykti po 12.5% visų pėsčiųjų.

Šios simuliacijos statistiniai duomenys bus pateikti apačioje pateiktuose paveikslėliuose ir bendri statistikos duomenys bus pateikti tekstu. Būtina atkreipti dėmesį, kad į statistiką nebuvo įtraukti duomenys, kai transporto priemonės arba pėstieji neturėjo laukti, kad atliktų savo veiksmą (statistiniai duomenys, be laukimo laiko 0 sekundžių):

- Simuliacijos metu atvyko 694 transporto priemonės, apytiksliai turėtų atvykti apie 720 = 3600 / 5 (SIMULATION_TIME / ARRIVAL_MEAN).
- Simuliacijos metu transporto priemonė vidutiniškai laukė apie 15.71 sekundžių.
- Simuliacijos metu atvyko 351 pėsčiasis, apytiksliai turėtų atvykti apie 360 = 3600 / 10 (SIMULATION_TIME / PED_ARRIVAL_MEAN).
- Simuliacijos metu pėsčiasis vidutiniškai laukė apie 14.67 sekundžių.

Tai yra bendri simuliacijos statistikos duomenys, norint apžvelgti detaliau, reikia žiūrėti, kas vyksta kiekvieno maršruto ir perėjos viduje. Tam atlikti buvo sukurtos diagramos, kur yra sukaupta informacija: kiek tam tikras objektas turėjo laukti ir kiek jų buvo. 10 paveikslėlyje yra pateikta diagrama, kiek vidutiniškai laukia kiekvienas pėsčiasis, kiekvienoje pėsčiųjų p erėjų poroje. Tuo tarpu 11 paveikslėlyje yra pateikta histograma, kiek bendrai laukė pėsčiųjų tam tikruose laukimo laiko tarpuose, šios diagramos pagalba galima pamatyti, kokie laukimo laiko tarpai dažniausiai pasikartoja. Iš pateiktos diagramos ir histogramos galima matyti, kad vidutiniškai kiekvienoje pėsčiųjų perėjoje pėstieji vidutiniškai laukia nuo 10.17s. iki 19.02s., geriausias atvejis yra [1, 2] ir [2, 1] pėsčiųjų perėjoje, o blogiausias [2, 3] ir [3, 2] pėsčiųjų perėjoje. Lyginant pėsčiųjų perėjų diagramų duomenis su bendros statistikos duomenimis, galime teigti jog jokių neigiamų atvejų nėra, nes didžioji dalis kitų pėsčiųjų perėjų vidurkiai yra labai arti bendro vidurkio 14.67s.

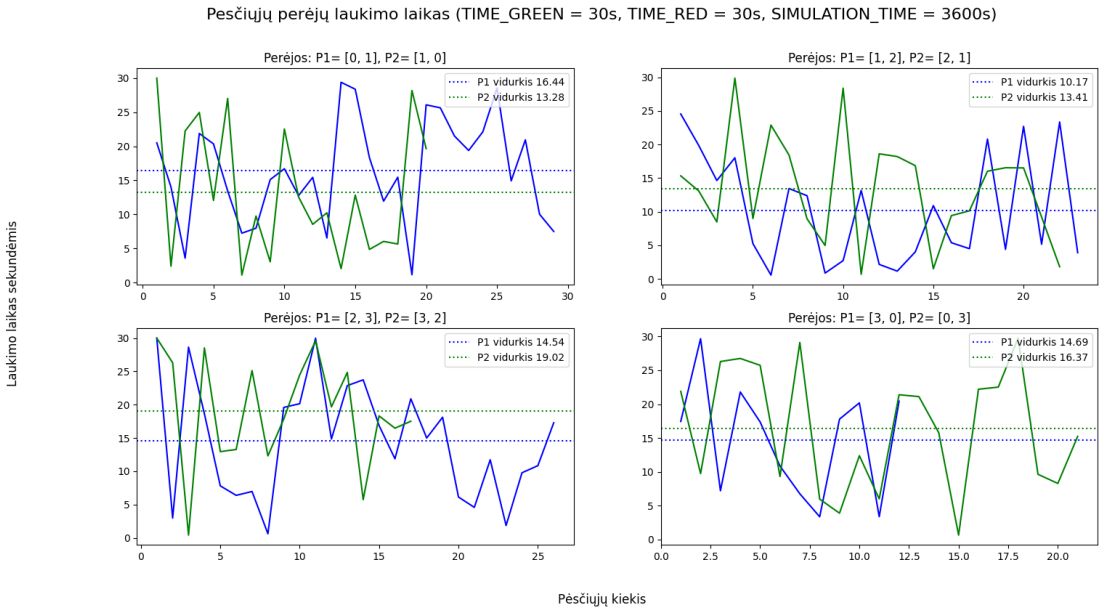
Šiam simuliacijos aplinkos prototipui svarbiausi elementai yra transporto priemonės ir jų laukimo laikas, nes pėsčiųjų eilės yra lengviau valdomos t. y. pėstieji vienu metu gali visi judėti ir jie užima mažiau vietos, tuo tarpu transporto priemonių eilės gali sudaryti problemas ir kituose keliuose ar sankryžose, jeigu jos susidaro per ilgos. Todėl reikia pažiūrėti kokius gauname statistinius duomenis iš diagramų, kurios yra pateiktos 12 paveikslėlyje ir transporto priemonių laukimo laiko histogramos 13 paveikslėlyje.

Iš pateiktos diagramos ir histogramos galima matyti, jog maršrutai 0: [5, 7, 3] ir 4: [1, 3, 7] yra pagrindinio kelio, kuris gauna 70% visos sankryžos transporto priemonių, o likęs kelias gauna tik 30% transporto priemonių srauto. Transporto priemonės maršrutuose vidutiniškai laukia nuo 14.89s. iki 16.22s., geriausias atvejis yra 6: [3, 5, 1] maršrute, o blogiausias 0: [5, 7, 3] maršrute.

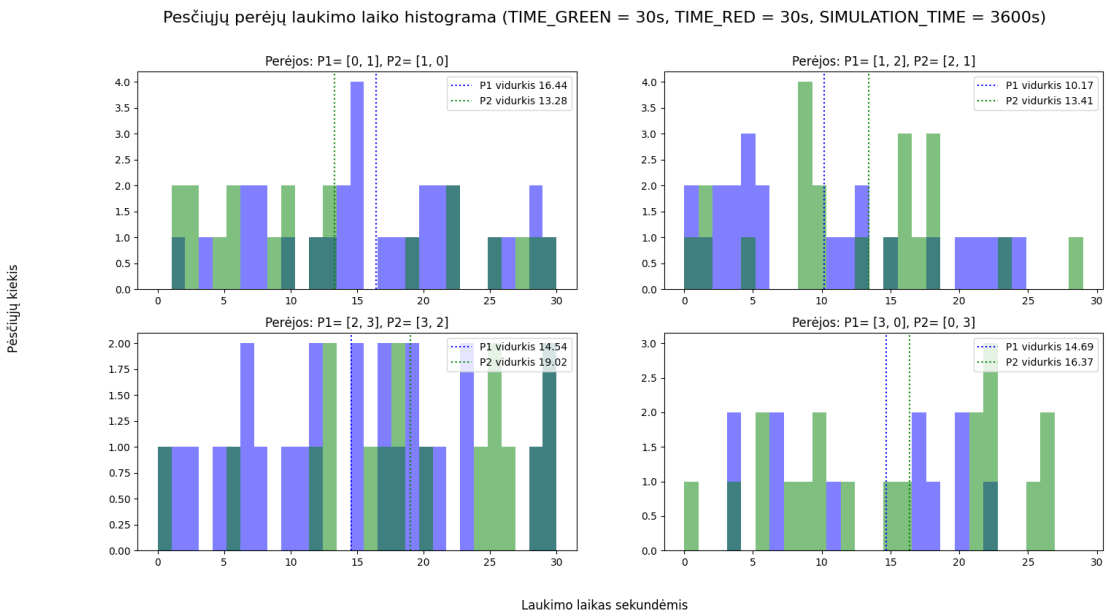
Lyginant transporto priemonių maršrutų diagramų duomenis su bendros statistikos duomenimis, galime teigti jog jokių neigiamų atvejų nėra, nes maršrutų vidurkiai yra labai arti bendro vidurkio 15.71s.

Pradinio varianto šviesoforo raudona ir žalia šviesa šviečia po 30s., todėl transporto priemonių ir pėsčiųjų vidutinis laukimo laikai nėra tokie skirtingi nuo jų bendros statistikos vidurkių. Tik dvi pėsčiųjų perėjos turėjo didesnius skirtumus tarp vidurkio, dėl simuliacijos pėsčiųjų mažesnio kiekio ir simuliacijos laiko, nes su daugiau duomenų šie skirtumai išnyktų ir priartėtų prie bendros statistikos vidurkių.

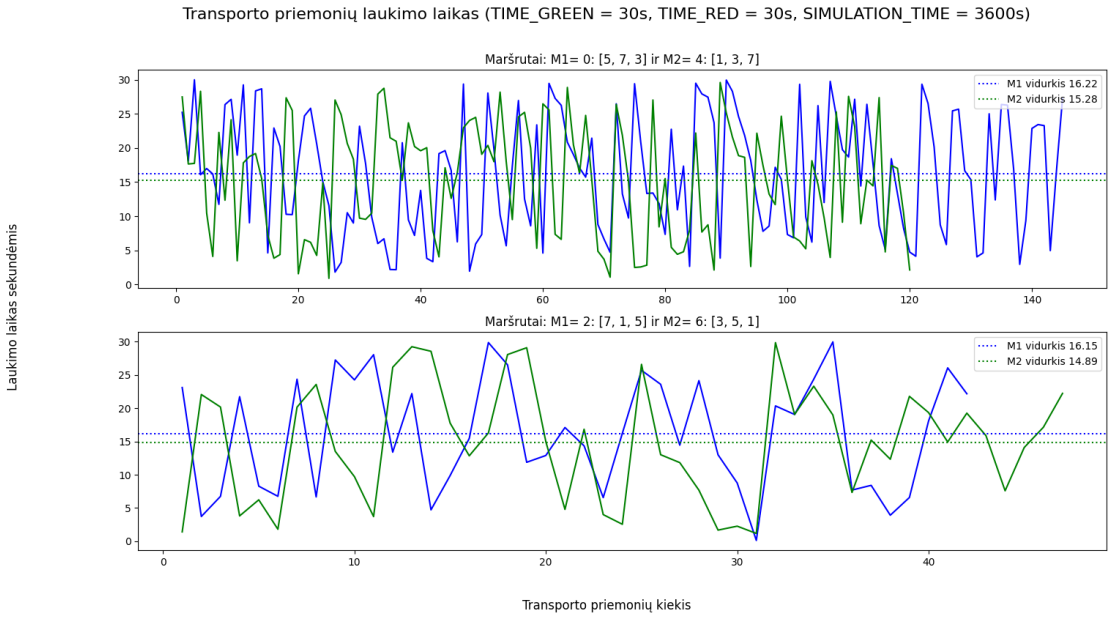
Norint pagerinti šios simuliacijos sankryžą, bus parodyti du skirtingi prototipų variantai, kai viename bus pakeista tik šviesoforo šviesų laiko tarpai, o kitame bus pridėti papildomi transporto priemonių maršrutai norint sumažinti vidutinį laukimo laiką. Galima pastebėti, jog pirmas variantas yra lengvesnis ir lengviau įgyvendamas, nes ne visada įmanoma tiesiog praplėsti kelius, tačiau tai duotų papildoma variantą problemos sprendimui.



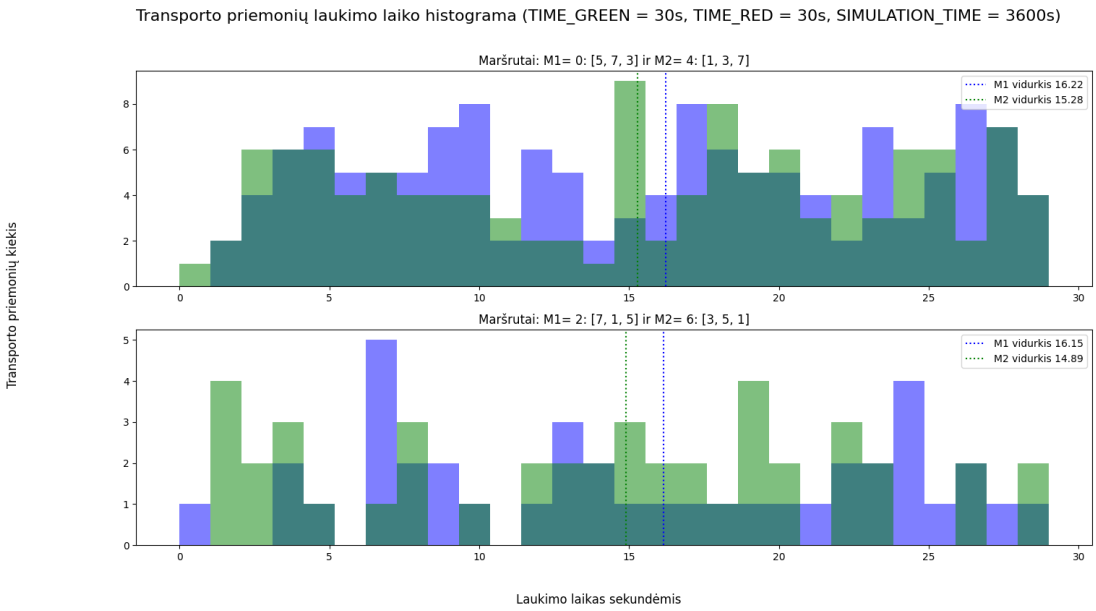
10 pav. Pradinio varianto pėsčiųjų perėjų laukimo laiko diagrama



11 pav. Pradinio varianto pėsčiųjų perėjų laukimo laiko histograma



12 pav. Pradinio varianto transporto priemonių maršrutų laukimo laiko diagrama



13 pav. Pradinio varianto transporto priemonių maršrutų laukimo histograma

Pirmas prototipo pagerinimo variantas, kai tik pakeičiamos tik šviesoforo šviesų laiko tarpai. Kartu reikia paminėti, jog kaip sukurtas šis prototipas, žalios šviesos kintamasis yra priskiriamas 0 grupei, kuri šiuo atveju yra pagrindiniai maršrutai 0 ir 4, o raudonos šviesos kintamas yra priskiriamas 1 grupei, kuriai priklauso 2 ir 6 maršrutai. Šiam variantui buvo pasirinkta, jog žalia šviesa šviečia 30 sekundžių, o raudona šviečia 20 sekundžių. Lyginant su pradiniu variantu visas šviesoforo ciklas trunka 50 sekundžių vietoj 60 sekundžių, tačiau rezultatai yra geresni, negu naudojant 30 sekundžių raudonai ir žalioms šviesoms. Taip nutiko, nes pagrindinio kelio maršrutuose 0 ir 4 važiuoja daugiau transporto priemonių, todėl joms reikia laukti mažiau laiko per raudoną šviesą, nes raudona šviečia tik 20 sekundžių vietoj 30 sekundžių. Šios simuliacijos aplinkos pakeisti kintamieji yra pateikti 13 lentelėje, o visi kiti yra lygiai tokie patys kaip 12 lentelėje.

13 lentelė. Simuliacijos aplinkos pirmo prototipo varianto pakeisti kintamieji

Kintamasis	**Paaiškinimas**
TIME_GREEN = 30	Simuliacijos aplinkos šviesoforo žalios šviesos būsenos laikas.
TIME_RED = 20	Simuliacijos aplinkos šviesoforo raudonos šviesos būsenos laikas.

Šios simuliacijos varianto statistiniai duomenys bus pateikti apačioje pateiktuose paveikslėliuose ir bendri statistikos duomenys bus pateikti tekstu. Būtina atkreipti dėmesį, kad į statistiką nebuvo įtraukti duomenys, kai transporto priemonės arba pėstieji neturėjo laukti, kad atliktų savo veiksmą (statistiniai duomenys, be laukimo laiko 0 sekundžių):

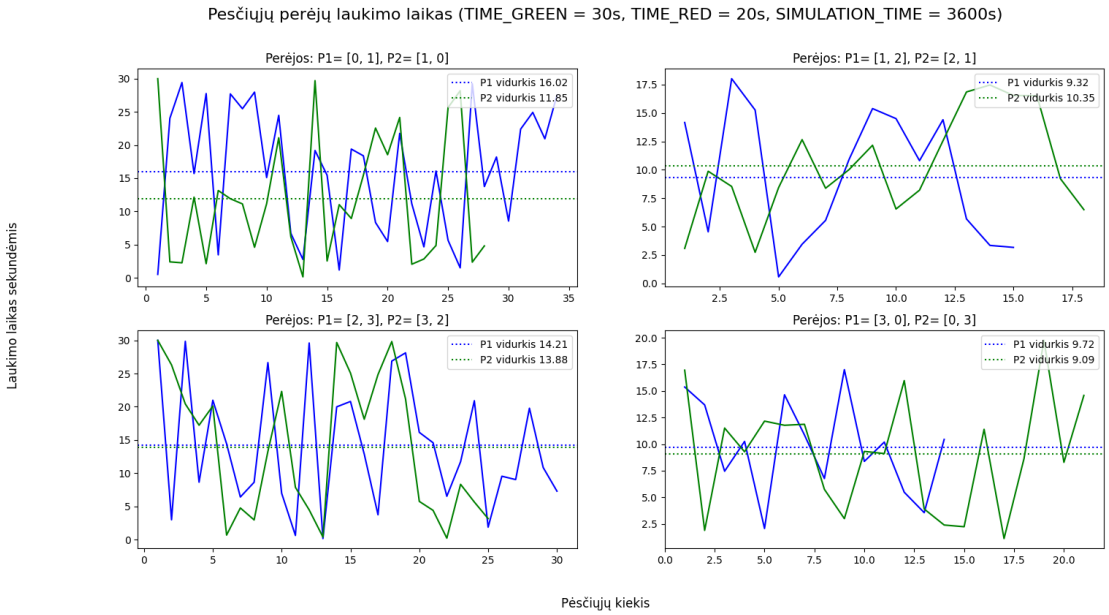
- Simuliacijos metu atvyko 694 transporto priemonės, apytiksliai turėtų atvykti apie 720 = 3600 / 5 (SIMULATION_TIME / ARRIVAL_MEAN).
- Simuliacijos metu transporto priemonė vidutiniškai laukė apie 12.56 sekundžių.
- Simuliacijos metu atvyko 351 pėsčiasis, apytiksliai turėtų atvykti apie 360 = 3600 / 10 (SIMULATION_TIME / PED_ARRIVAL_MEAN).
- Simuliacijos metu pėsčiasis vidutiniškai laukė apie 12.45 sekundžių.

Lyginant su pradiniu prototipu vidutinis transporto priemonės laukimo laikas sumažėjo iš 15.71 sekundžių į 12.56 sekundžių, o pėsčiųjų vidutinis laukimo laikas sumažėjo iš 14.67 sekundžių į 12.45 sekundžių. 14 paveikslėlyje yra pateikta pėsčiųjų perėjų laukimo laikų diagrama, o 15 paveikslėlyje yra pateikta pėsčiųjų perėjos laukimo laikų histograma ir šių diagramų duomenų galima pastebėti, jog pėsčiųjų perėjos [0, 1], [1, 0], [2, 3] ir [3, 2] turi didesnius laukimo laikus, negu likusios pėsčiųjų perėjos. Taip yra, nes šioms pėsčiųjų perėjoms šviečia raudona šviesa 30s., o likusioms tik 20s., todėl pėstieji vidutiniškai laukia nuo 9.09s. iki 16.02s., geriausias atvejis yra [3, 0] ir [0, 3] pėsčiųjų perėjoje (šviečia raudona 20s.), o blogiausias atvejis yra [0, 1] ir [1, 0] pėsčiųjų perėjoje (šviečia raudona 30s.). Lyginant pėsčiųjų perėjų diagramų duomenis su bendros statistikos duomenimis galima teigti, jog šio prototipo varianto pėsčiųjų laukimo laikas sumažėjo, dėl to, nes šviesoforo pilnas ciklas sumažėjo iš 60s. į 50s., kas sumažina vidutinį laukimo laiką toms pėsčiųjų perėjos, kurios laukia mažiau. Todėl yra daug daugiau pėsčiųjų perėjų, kurios pėstieji vidutiniškai laukia daug mažiau negu bendro laukimo vidurkio 12.45s.

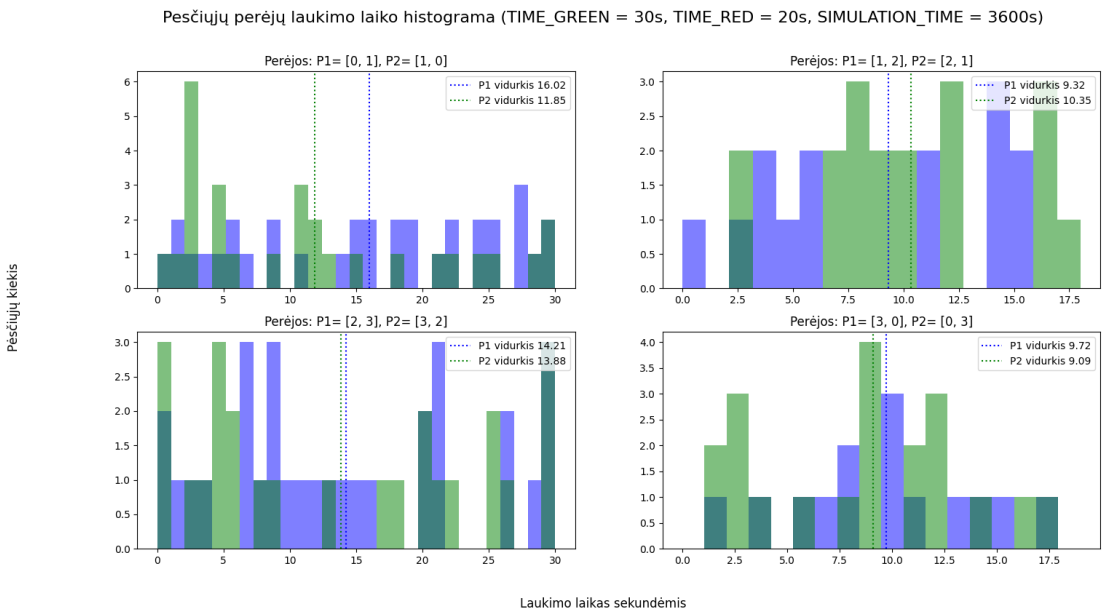
Pėsčiųjų perėjų laukimo laikai nėra tokie svarbūs kaip transporto priemonių laukimo laikai, todėl reikia pažiūrėti kas vyksta transporto priemonių maršrutuose. 16 paveikslėlyje yra pateikta transporto priemonių laukimo laikų diagrama, o 17 paveikslėlyje yra pateikta transporto priemonių laukimo laikų histograma. Histogramos pagalba galima pastebėti, kad pagrindinio kelio maršrutai, kuriuose turi laukti tik 20s. per raudona šviesa, nėra nei vieno atvejo, kad vairuotojai lauktų ilgiau negu 20s. tuo tarpu nepagrindinio kelio maršrutai, kurie turi laukti 30s. per raudona šviesą, turi atvejų, kai vairuotojai laukia ilgiau negu 20s. Todėl transporto priemonių vairuotojai vidutiniškai laukia nuo 10.51s. iki 17.35s., geriausias atvejis yra 4: [1, 3, 7] maršrute (šviečia raudona 20s.), o

blogiausias atvejis yra 6: [3, 5, 1] maršrute (šviečia raudona 30s.). Lyginant transporto priemonių diagramų duomenis su bendros statistikos duomenis galime teigti, jog šio prototipo varianto vairuotojų laukimo laikas sumažėjo, nes didžioji dalis visų transporto priemonių turi laukti mažiau šalia raudonos šviesos, tai padeda sumažinti sankryžos vidutinį transporto priemonių laukimo laiką. Todėl pagrindinio kelio maršrutų vidutiniai laukimo laikai yra žemesni negu bendras laukimo vidurkis, o ne pagrindinio kelio maršrutų vidutiniai laukimo laikai yra didesni negu bendras laukimo vidurkis 12.56s.

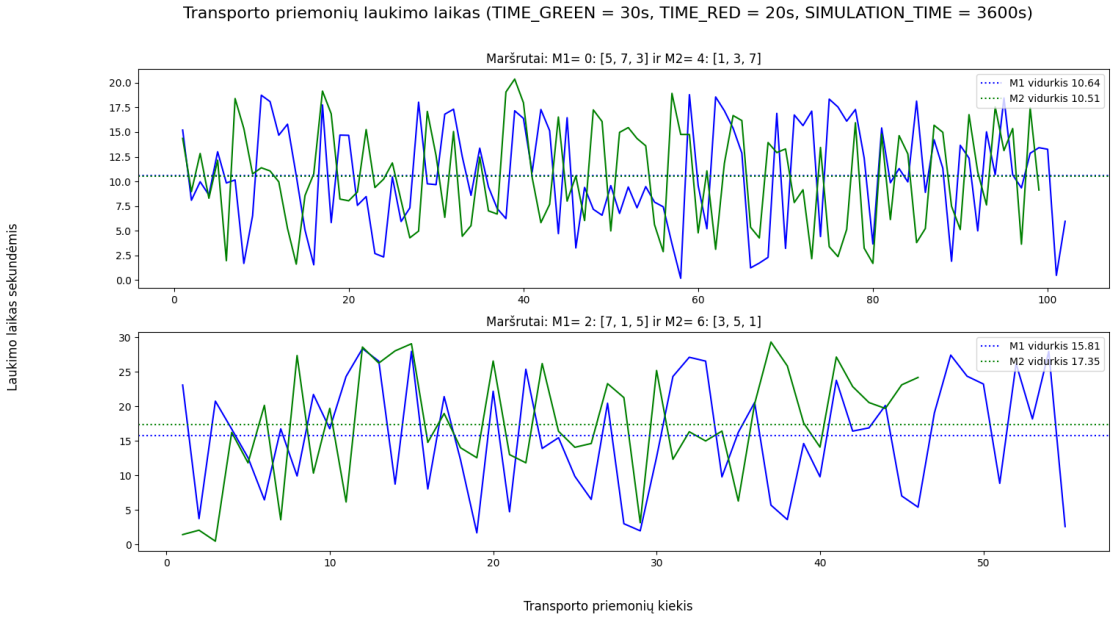
Apibendrinant šį prototipo variantą, galime matyti, kad šis variantas yra geresnis negu pradinis variantas, kuriame šviesoforas šviečia 30 sekundžių žaliai ir raudonai. Transporto priemonių vidutinis laukimo laikas sumažėjo iš 15.71 sekundžių į 12.56 sekundžių, o pėsčiųjų vidutinis laukimo laikas sumažėjo iš 14.67 sekundžių į 12.45 sekundžių. Tačiau iš transporto priemonių diagramų buvo pastebėta, jog nepagrindiniuose kelių maršrutuose susidaro ilgesnės transporto priemonių eilės negu pradiniame variante, kas gali turėti neigiamos įtakos jeigu susidarytų per ilgos eilės. Todėl šviesoforo šviesos pakeitimai, kad žalia šviestų 30 sekundžių, o raudona šviestų 20 sekundžių turi gerą įtaką tokiose sankryžose, kur tikimasi daug daugiau transporto priemonių pagrindiniuose keliuose.



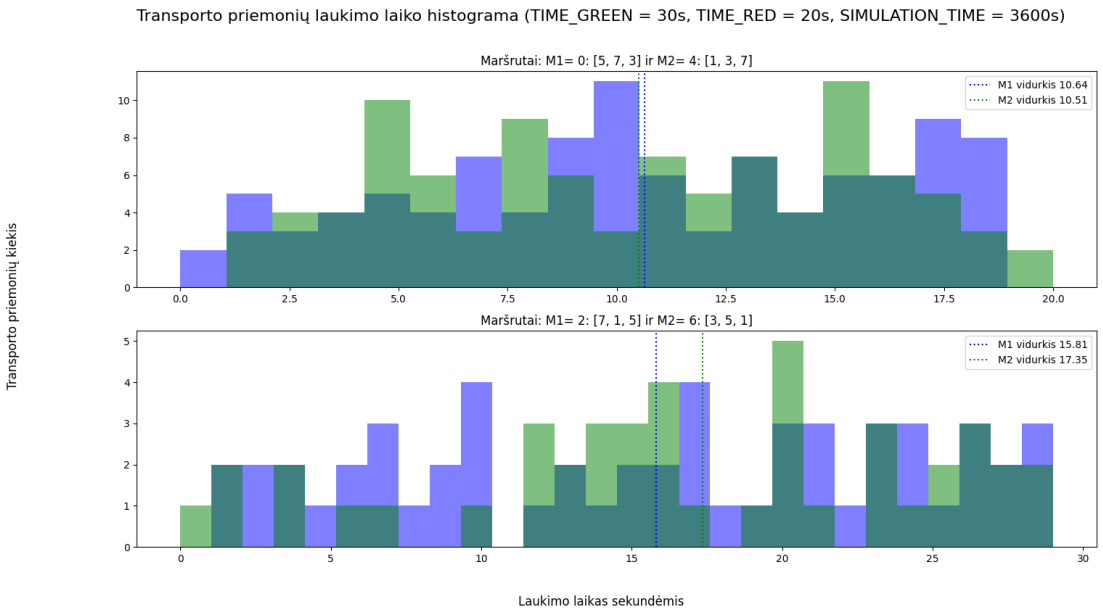
14 pav. Pirmo varianto pėsčiųjų perėjų laukimo laiko diagrama



15 pav. Pirmo varianto pėsčiųjų perėjų laukimo laiko histograma

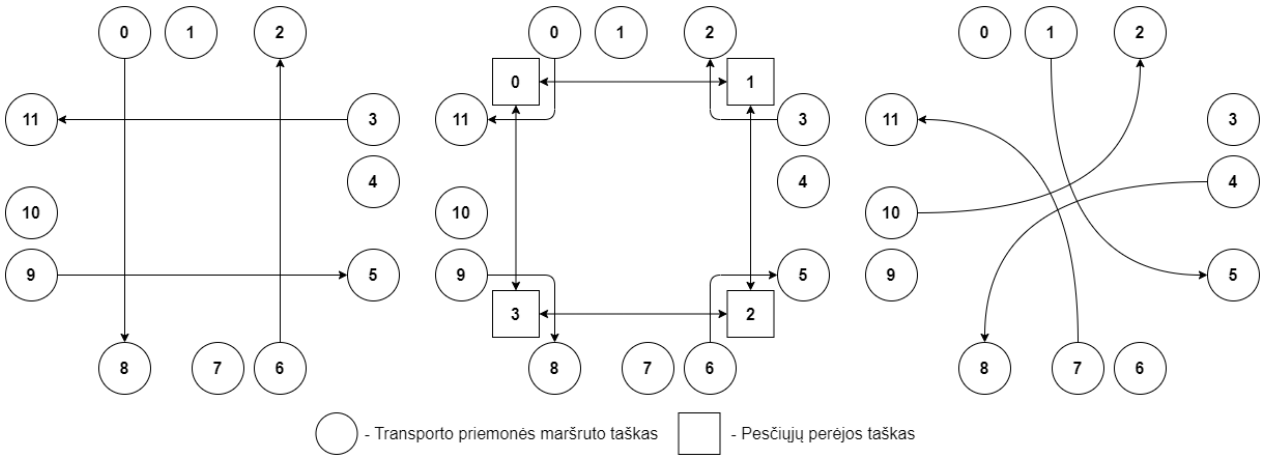


16 pav. Pirmo varianto transporto priemonių maršrutų laukimo laiko diagrama



17 pav. Pirmo varianto transporto priemonių maršrutų laukimo laiko histograma

Antras prototipo pagerinimo variantas yra ta pati sankryžą tačiau važiavimo kelias išsiplečia į dvi kelio linijas, kairioji kelio linija galima važiuoti tik į kairę, o dešiniąją galima važiuoti tiesiai ir į dešinę. Pagerintos sankryžos antro varianto nukreipiamojo grafo diagrama yra pateikta 18 paveikslėlyje, o simuliacijos aplinkos pakeisti kintamieji pateikti 14 lentelėje.



18 pav. Sankryžos nukreipiamo grafo diagrama pagal pagerinimo 2 variantą

14 lentelė. Simuliacijos aplinkos antrojo prototipo varianto pakeisti kintamieji

Kintamasis	**Paaiškinimas**
Links = {0: [8, 11], 3: [11, 2], 6: [2, 5], 9: [5, 8], 1: [5], 4: [8], 7: [11], 10: [2]}	Simuliacijos aplinkos nukreipiamasis grafas, kuriame matoma kokie yra transporto priemonių maršrutai ir kokie galimi maršruto išvažiavimai. Šis kintamasis sukurtas iš 18 pav.
LinkGroups = [0, 1, 0, 1, 0, 1, 0, 1]	Simuliacijos aplinkos grupių sąrašas, kurio pagalba šviesoforas gali valdyti visus kelius vienu metu.
TurnTimes = [[T_DEPART_MIDDLE, T_DEPART_RIGHT], [T_DEPART_LEFT]]	Simuliacijos aplinkos posūkių atlikimo sąrašas su masyvais, pirmasis masyvo elementas yra atsakingas už posūkius: tiesiai ir į dešinę, o kitas elementas atsakingas už posūkius į kairę.
RoadWeights = [0.28, 0.12, 0.28, 0.12, 0.07, 0.03, 0.07, 0.03]	Simuliacijos aplinkos tikimybės transporto priemonėms atvažiuoti į tam tikrą transporto priemonių maršrutą.
DestinationWeights = [[0.50, 0.50], [1]]	Simuliacijos aplinkos tikimybių sąrašas į kurį posūkį suktų, pirmas masyvo elementas atsakingas už posūkius: tiesiai ir į dešinę, o kitas masyvo elementas atsakingas už posūkį į kairę.

Pradinis simuliacijos aplinkos prototipas turėjo tikimybių sąrašą, kur gali atsirasti transporto priemonė, taip buvo galima nurodyti, kur daugiau atvyksta transporto priemonių ir kurį maršrutą jos pasirenka, tuo pačiu turėjo tikimybių sąrašą kokį posūkį su kokia tikimybę atliks. Todėl, kad pasikeitė visas nukreipiamasis grafas *Links* reikėjo pakeisti ir du kintamuosius *RoadWeights* ir *DestinationWeights*, pradinio prototipo *RoadWeights* sudarė šios tikimybės: 0.35, 0.15, 0.35 ir 0.15, o posūkių tikimybes *DestinationWeights* sudarė šios tikimybės: 0.40, 0.40 ir 0.20. Iš šių kintamųjų galime matyti jog transporto priemonių maršrutai 0 ir 4 turėjo po 35% tikimybes, kad juose atsiras transporto priemonė, o maršrutai 2 ir 6 turi po 15%. Tuomet iš kiekvieno maršruto buvo galima važiuoti į visas puses: tiesiai su 40%, į dešinę su 40% ir į kairę su 20% tikimybėmis. Norint išlaikyti lygiai tokias pačias tikimybes, šiam prototipo variantui buvo sukurti nauji

RoadWeights ir *DestinationWeights* kintamieji, kad atitiktų pradinį prototipą. Naujame prototipe galimų maršrutų kiekis padvigubėjo iš 4 į 8, kitaip sakant kiekvienas kelias gavo po naują kelio liniją iš kurios galima važiuoti tik į kairę, o senos linijos negalima važiuoti į kairę. Todėl buvo dauginamos maršrutų pasirinkimo tikimybės su posūkio pasirinkimo tikimybėmis, nes transporto priemonė gali pasirinkti iškarto ar nori važiuoti į kairę ar tiesiai ir į dešinę. Atlikti skaičiavimai:

- 0.35 * (0.4 + 0.4) = 0.28 ir 0.35 * 0.2 = 0.07, taip gauname naujas tikimybes iš pradinio prototipo tikimybių. Šiuo atveju transporto priemonė gali pasirinkti maršrutą, kuriuo galima važiuoti tik į kairę su 7% tikimybę arba maršrutą, kuriuo galima važiuoti tiesiai ir į dešinę su 28% tikimybę.
- 0.15 * (0.4 + 0.4) = 0.12 ir 0.15 * 0.2 = 0.03, taip gauname naujas tikimybes iš pradinio prototipo tikimybių. Šiuo atveju transporto priemonė gali pasirinkti maršrutą, kuriuo galima važiuoti tik į kairę su 3% tikimybę arba maršrutą, kuriuo galima važiuoti tiesiai ir į dešinę su 12% tikimybę.
- Posūkio pasirinkimo tikimybės buvo atskirtos tarp dviejų maršrutų dėl to, kad tikimybė važiuoti tiesiai ir į dešinę buvo vienodos, transporto priemonė gali pasirinkti važiuoti tiesiai arba į dešinė su 50% tikimybe. O transporto priemonė, kuri atvyko į kairės posūkio maršrutą, gali pasukti tik į kairę su 100% tikimybe.

Šios simuliacijos varianto statistiniai duomenys bus pateikti apačioje pateiktuose paveikslėliuose ir bendri statistikos duomenys bus pateikti tekstu. Būtina atkreipti dėmesį, kad į statistiką nebuvo įtraukti duomenys, kai transporto priemonės arba pėstieji neturėjo laukti, kad atliktų savo veiksmą (statistiniai duomenys, be laukimo laiko 0 sekundžių)::

- Simuliacijos metu atvyko 694 transporto priemonės, apytiksliai turėtų atvykti apie 720 = 3600 / 5 (SIMULATION_TIME / ARRIVAL_MEAN).
- Simuliacijos metu transporto priemonė vidutiniškai laukė apie 16.07 sekundžių.
- Simuliacijos metu atvyko 351 pėsčiasis, apytiksliai turėtų atvykti apie 360 = 3600 / 10 (SIMULATION_TIME / PED_ARRIVAL_MEAN).
- Simuliacijos metu pėsčiasis vidutiniškai laukė apie 14.67 sekundžių.

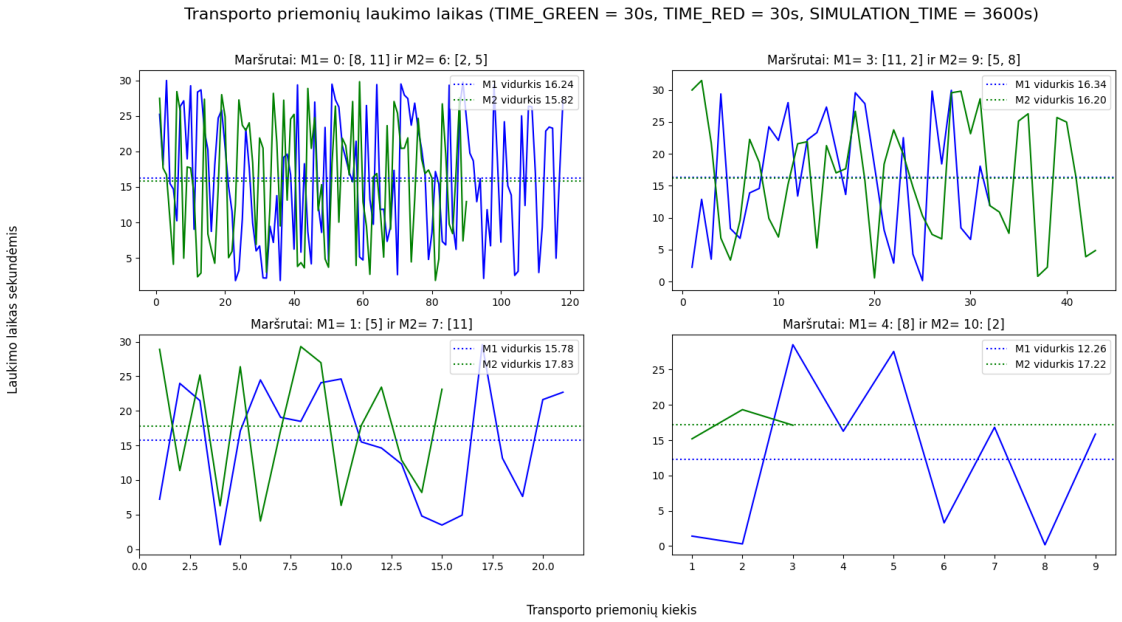
Lyginant su pradiniu prototipo variantu, vidutinis laukimo laikas transporto priemonėms pablogėjo iš 15.71s į 16.07s., skirtumas tarp šių vidurkių nėra toks didelis, bet galime matyti jog laukimo laikas išaugo. Tačiau šiame variante 4 papildomi maršrutai, kuriais galima sukti tik į kairę,

o likę maršrutai rūpinasi tik tiesiai ir į dešinę važiuojančiomis transporto priemonėmis, kitaip sakant galimas dvigubas transporto priemonių srautas kiekviename kelyje, ypač, jeigu transporto priemonės važiuotų dažniau sukdami į kairę pusę. Tuo tarpu pėsčiųjų perėjos vidutinis laukimo laikas liko toks pats, kaip pradinio prototipo varianto, nes šiame variante, buvo pakeisti tik transporto priemonių maršrutai, kas neliečia pėsčiųjų perėjų, nes pėstieji tiek pat turi laukti leidimo eiti. Todėl, kad pėsčiųjų perėjų diagrama ir histograma yra vienodos į jas nebus kreipta dėmesio, nes viskas analogiška pradinio prototipo varianto pėsčiųjų perėjų diagramoms ir bendrų statistikos duomenims.

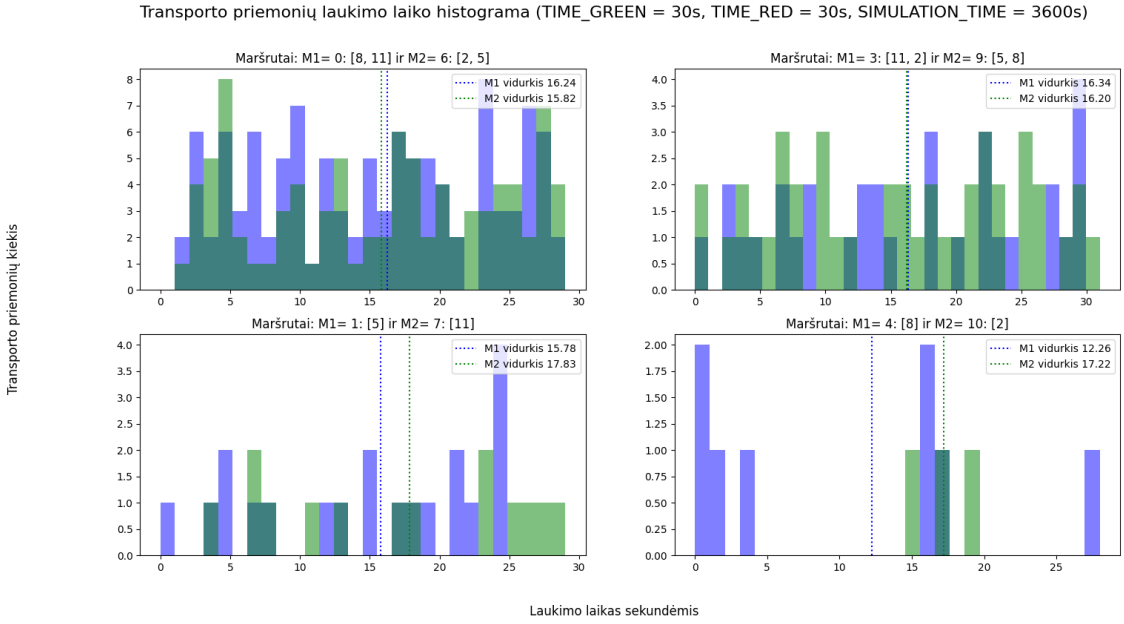
Transporto priemonių maršrutų diagrama yra pateikta 19 paveikslėlyje, o histograma yra pateikta 20 paveikslėlyje. Iš šių abiejų diagramų galima pastebėti, jog simuliacijoje yra papildomi 4 maršrutai, kurie rūpinasi tik posūkiu į kairę pusę, kas padeda sumažinti transporto priemonių srovę kitiems maršrutams. Tuo pačiu tai būtų geresnis variantas tiems vairuotojams, kurie nori važiuoti tik į kairę, nes ten būtų mažesnė eilė ir kartu sumažinama galimos eilės kituose maršrutuose. Iš diagramos ir histogramos galime matyti, jog transporto priemonės vidutiniškai laukia nuo 12.26s. iki 17.83s., geriausias atvejis yra naujame kairės posūkio maršrute 4: [8], o blogiausias atvejis yra irgi naujame kairės posūkio maršrute 7: [11]. Nors bendros statistikos vidutinis laukimo laikas transporto priemonėms truputį išaugo, galime pastebėti, jog senuose maršrutuose vidutinis laukimo laikas liko beveik toks pats ir dalis transporto priemonių buvo perkelta iš šių maršrutų į naujus maršrutus, tai gali turėti įtakos, kai atvažiuotų daugiau negu 694 transporto priemonės per tą patį simuliacijos laiko tarpą.

Šis pagerinimo variantas pagal statistinius duomenis yra beveik toks pats, kaip pradinis prototipas, tačiau šis variantas turi privalumų, kad galimas dvigubas transporto priemonių srautas ir šis variantas būtų geresnis, jeigu važiuotų daugiau transporto priemonių naujais maršrutais. Naujas pagerinimo variantas nėra visai tinkamas ir galimas tokioms sankryžoms, kurios neturi galimybės arba vietos sukurti naujiems kelių linijoms, kur būtų šie nauji maršrutai. Tai pat tai būtų daug brangesnis variantas negu pirmas pagerinimo variantas, nes čia reikėtų sankryžos fizinio pakeitimo.

Norint įsitinkinti šios sankryžos privalumus, reikėtų atlikti kiekvienos sankryžos streso testą, kas simuliuotų piko valandą. Todėl bus kartu simuliuojami visi prototipo variantai, kai transporto priemonės ir pėstieji atvyksta 5 kartus greičiau, negu įprastai.



19 pav. Antro varianto transporto priemonių maršrutų laukimo laiko diagrama



20 pav. Antro varianto transporto priemonių maršrutų laukimo laiko histograma

Streso testas simuliuoja piko valandos pavyzdį visuose trijuose prototipo variantuose, pakeisti kintamieji yra pateikti 15 lentelėje. Atsižvelgus į pakeistus kintamuosius galima pastebėti, kad transporto priemonių ir pėsčiųjų kiekis bus 5 kartus didesnis negu įprastuose simuliacijos atvejuose. Visi pagrindiniai statistikos duomenys bus pateikti 16 lentelėje, kurioje bus palyginta įprastų prototipų variantų duomenys su to pačio prototipo streso testo simuliacijos prototipo

duomenimis. Galiausiai daugiau dėmesio bus kreipta į tuos duomenis ir prototipus, kur streso testo metu bus pamatytos neigiamos įtakos (susidaro didelės eilės, ilgiau laukia ir panašiai).

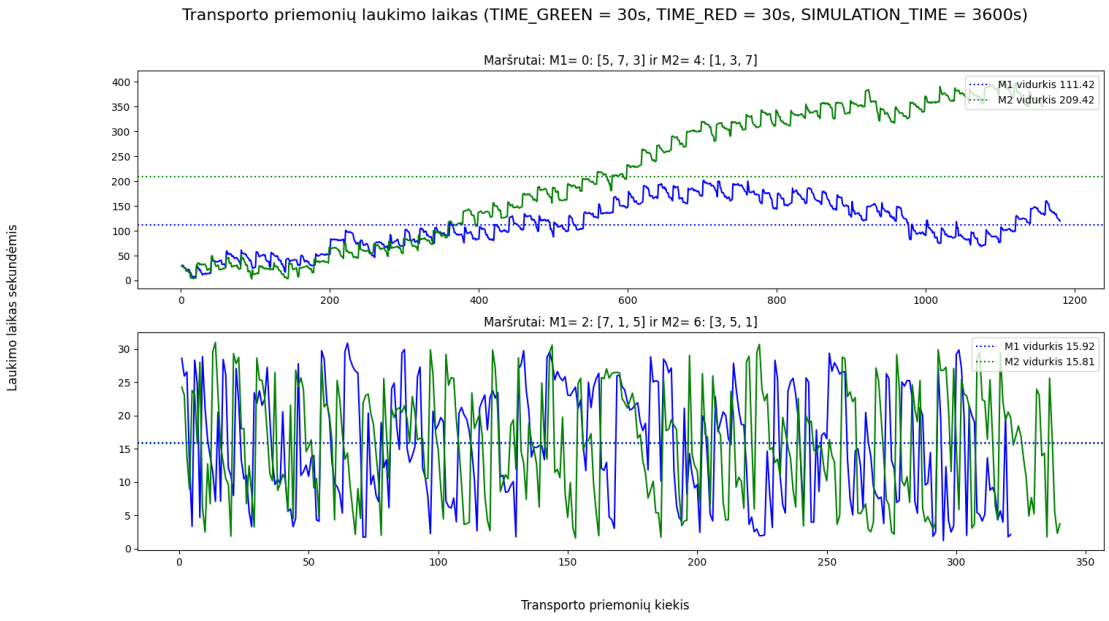
15 lentelė. Simuliacijos aplinkos streso testų pakeisti kintamieji

Kintamasis	**Paaiškinimas**
ARRIVAL_MEAN = 1	Kintamasis kuris parodo, kas kiek sekundžių vidutiniškai turėtų būti sukurta nauja transporto priemonė. Klasė *CarGenerator* naudoja Puasono pasiskirstymą.
PED_ARRIVAL_MEAN = 2	Kintamasis kuris parodo, kas kiek sekundžių vidutiniškai turėtų būti sukurtas naujas pėsčiasis. Klasė Ped*Generator* naudoja Puasono pasiskirstymą.

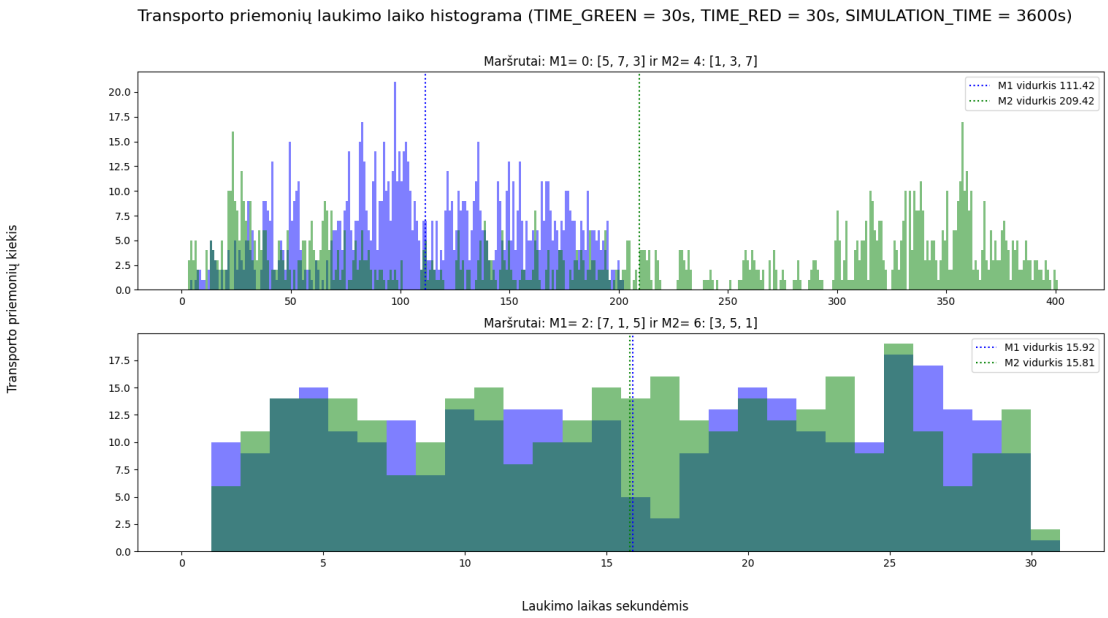
16 lentelė. Simuliacijos prototipų duomenų palyginimas su streso testo duomenimis

Simuliacijos prototipo pavadinimas	**Transporto priemonių kiekis**	**Vidutinis laukimo laikas**	**Pėsčiųjų kiekis**	**Vidutinis laukimo laikas**
Pradinis variantas	694	15.71 s.	351	14.67 s.
Pradinio varianto streso testas	3625	128.20 s.	1821	15.58 s.
Pirmas variantas	694	12.56 s.	351	12.45 s.
Pirmo varianto streso testas	3625	17.34 s.	1821	12.94 s.
Antras variantas	694	16.07 s.	351	14.67 s.
Antro varianto streso testas	3625	19.83 s.	1821	15.58 s.

Išanalizavus lentelėje pateiktus duomenis galime, pamatyti, jog streso testo metu pradinis prototipo variantas turi problemą lyginant su pirmuoju ir antruoju prototipų variantu. Pradiniame streso testo variante transporto priemonės vidutiniškai laukia 128.20s. lyginant su pirmo varianto streso testo 17.34s. ir antro varianto streso testo 19.83s. vidutiniais laikais. Galima pastebėti, jog kažkas yra negerai būtent transporto priemonių maršrutuose, tam įsitikinti yra pateiktos šio streso testo diagrama 21 paveikslėlyje ir histograma 22 paveikslėlyje. Iš pateiktos diagramos galime matyti, kad pagrindinio kelio (70% visų transporto priemonių srauto) abu maršrutai turi labai didelį vidutinį laukimo laiką, kas nėra normalu lyginant su kitais streso testo variantais. Tuo pačiu šioje diagramoje matome, jog maršrutų 0: [5, 7, 3] ir 4: [1, 3, 7] laukimo laiko linijos auga, kas reiškia, kad transporto priemonių mažiau spėja išvažiuoti iš eilės, negu atvažiuoja naujų transporto priemonių, todėl šiuo atveju susidarytų transporto priemonių kamštis šiuose maršrutuose. Galiausiai galime pamatyti, jog kiti likę maršrutai neturi transporto priemonių kamščių, nes juose atvažiuoja tik 30% visų šios sankryžos transporto priemonių. Išanalizavus šio varianto histogramą, galime lengviau pastebėti, kad 4: [1, 3, 7] turi didesnes problemas, negu 0: [5, 7, 3] maršrutas, nes šiame maršrute yra atvejų, kai transporto priemonės dažniausiai laukia virš 350s., kad atliktų savo kelionę ir yra atvejų, kai laukia net ir virš 400s.



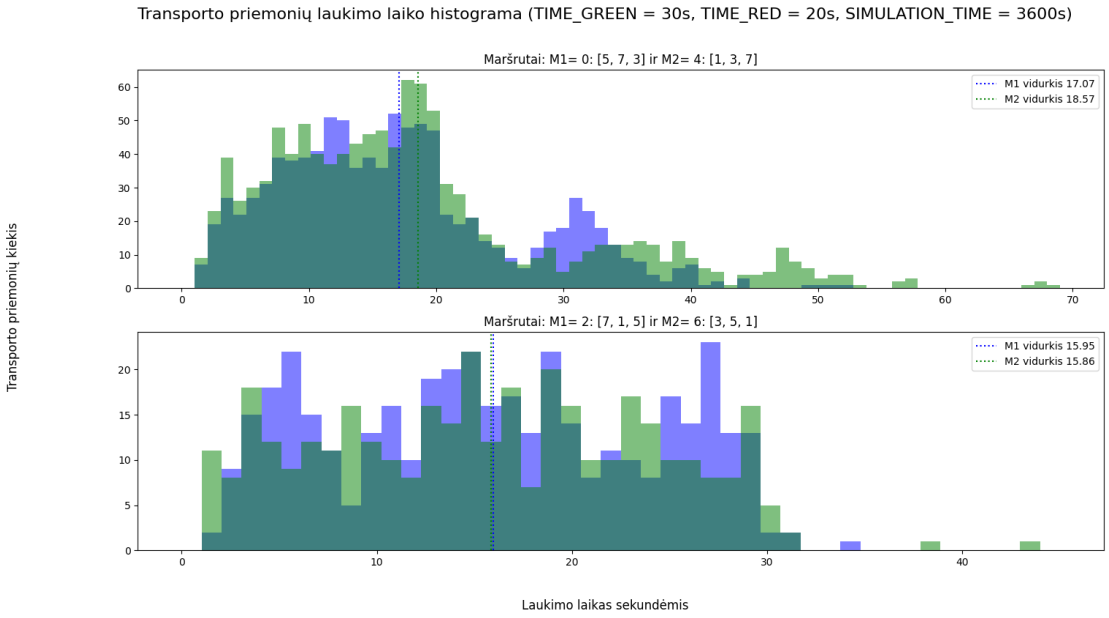
21 pav. Pradinio varianto streso testo transporto priemonių maršrutų laukimo laiko diagrama



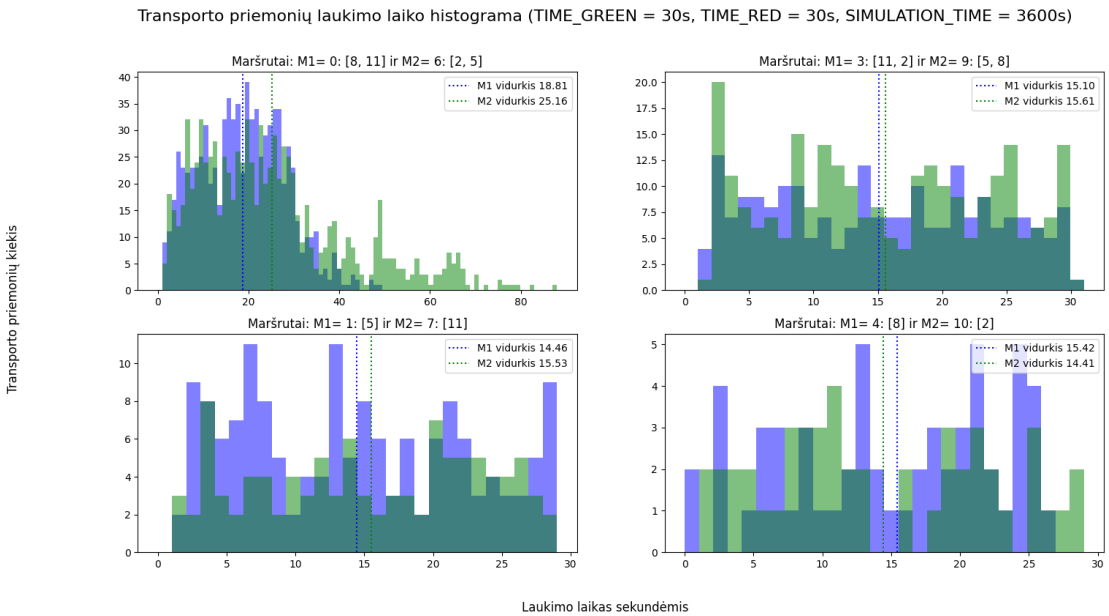
22 pav. Pradinio varianto streso testo transporto priemonių maršrutų laukimo laiko histograma

Atsižvelgus į 16 lentelės duomenis galima pastebėti, jog pirmo varianto streso testo ir antro varianto streso testo transporto priemonių laukimo laikai išaugo lyginant su šių variantų ne streso testo duomenimis. Pirmo varianto streso testo vidutinis laukimo laikas išaugo iki 17.34s. iš 12.56s., pirmo varianto streso testo transporto priemonių laukimo laikų histograma pateikta 23 paveikslėlyje, nes histogramoje lengviau pamatyti atvejus, kai transporto priemonės laukia ilgiau negu numatyta. Iš histogramos duomenų matoma, kad maršrutai 0: [5, 7, 3] ir 4: [1, 3, 7] turi daug atvejų, kai transporto priemonės laukia daugiau negu 20 sekundžių, kai šviečia raudona šviesa, todėl galima teigti, jog susidaro mažos eilės, tačiau tai neturi jokios blogos įtakos, kai pradinio varianto streso teste. Tuo tarpu kiti likę maršrutai tik retais atvejais laukia ilgiau negu 30 sekundžių, kai šviečia raudona šviesa. Antro varianto streso testo vidutinis laukimo laikas išaugo iki 19.83s. iš 16.07s., antro varianto streso testo transporto priemonių laukimo laikų histograma pateikta 24 paveikslėlyje. Iš histogramos duomenų matoma, kad tik pagrindinio kelio su didžiausiu srautu maršrutai turi problemų su laukimo laiku, šie maršrutai būtų: 0: [8, 11] ir 6: [2, 5], šiuose maršrutuose yra daug atvejų, kai transporto priemonės laukia daugiau negu 30 sekundžių, kai šviečia raudona šviesa. Todėl galima teigti, kad susidaro transporto priemonių eilės, tačiau atsižvelgus į visus kitus maršrutus, kur transporto priemonės laukia, viskas yra suvaldyta ir labai retai bus atvejis, kad susidarytų transporto priemonių eilės šiuose maršrutuose.

Išanalizavus streso testo duomenis galima pastebėti, kad pirmas ir antras variantas yra geresnis sprendimas negu pradinis variantas. Ypač jeigu atsitiktų toks atvejis kaip transporto priemonių spūstis ar piko valandos atvejis. Pirmas variantas yra pigesnis ir lengvesnis sprendimas, tuo tarpu antras variantas yra brangesnis, tačiau duoda lankstumo išspręsti ir kitas problemas.



23 pav. Pirmo varianto streso testo transporto priemonių maršrutų laukimo laiko histograma



24 pav. Antro varianto streso testo transporto priemonių maršrutų laukimo laiko histograma

4. Rezultatai ir išvados

Šiame darbe buvo atliktos dvi analizės, kaip prototipų kūrimas padeda sukurti informacinę sistemą ir kokią naudoja atneša prototipo kūrimas. Kitos analizės metu buvo analizuojamos diskretinių įvykių sistemos, kurios buvo sukurtos naudojantis „SimPy“ bibliotekos pagalba. Šių dviejų analizių pagalba buvo įsitikinta, kad naudojant „SimPy“ biblioteka galima sukurti lengviau ir greičiau simuliacijos prototipus, kurių pagalba galima patikrinti kritinius sistemos atvejus ar pakeitimus.

Analitinėje dalyje buvo analizuojama „SimPy“ biblioteka, kad butų įsitinkinta, jog ši biblioteka padės įgyvendinti pasirinktą diskretinių įvykio prototipo simuliaciją. Analizės metu buvo pastebėta, kad šios bibliotekos pagalba galima sukurti simuliacijos aplinkos prototipą, kuriame visi procesai ir įvykiai veikia toje pačioje aplinkoje, kas tiktų sankryžos prototipo simuliavimui.

Norint sukurti tam tikros sankryžos prototipą, buvo išsirinkta dviejų kelių susikirtimo sankryža. Ši sankryža buvo pasirinkta, nes ji yra įprasta sankryža, kuri turi visus reikalingus elementus: transporto priemonių keliai, pėsčiųjų perėjos keliai ir šviesoforai. Turint šiuos tris elementus prototipo variacijos gali būti keičiamos padidinant arba sumažinant transporto priemonių kelių kiekį, pėsčiųjų perėjos kelių kiekį arba modifikuojant šviesoforo šviesų ciklo laikus. Todėl sukurtas simuliacijos aplinkos prototipas yra lankstus ir galima sukurti keletą variantų, kad būtų galima pagerinti pradinį prototipą.

Atlikus simuliacijos aplinkos prototipą buvo sukurti 2 skirtingi pagerinti variantai. Pirmo varianto tikslas buvo pagerinti pradinį prototipą, pakeičiant tik šviesoforo šviesų ciklo laikus, tai padėtų pamatyti ar galima pagerinti tam tikras sankryžas naudojant tik šviesoforą kaip kintamą elementą. Antro varianto tikslas buvo pagerinti pradinį prototipą, pakeičiant pačių kelių struktūrą, tai padėtų pamatyti ar galima pagerinti sankryžą brangesniu būdu ir ar tai būtų geresnis sprendimas negu pirmas variantas. Verifikavimo dalyje atliktus statistine analizę buvo pastebėta, kad pradinio prototipo pagerinimui pirmas variantas yra geresnis sprendimas negu antras pagerinimo variantas. Tuo pačiu pirmas variantas yra pigiausias sprendimas, nes nereikia keisti pačios sankryžos.

LITERATŪROS SĄRAŠAS

[AK21] *Development of a prototype of a medical information system for a clinical diagnostic center.* **Andrikov, D.A. ir Kuchin, A.S.** 2021 m., Procedia Computer Science, Volume 186, p. 287-292.

[ASG15] **M, Allen, A, Spencer ir A, Gibson.** *Right cot, right place, right time: improving the design and organisation of neonatal care networks – a computer simulation study.* Anglija : NIHR Journals Library, 2015.

[BGS+14] *Optimal Run Length for Discrete-event Distributed Cluster-based Simulations.* **Borges, Francisco, et al.** 2014 m., Procedia Computer Science, Volume 29, p. 73-83.

[BMT21] **BMT.** Discrete event simulation. *Discrete event simulation.* [Tinkle] 2021 m. Gegužės 14 d. https://www.bmt.org/industries/defence-and-security/discrete-event-simulation/.

[Com20] **CompTIA.** IT Industry Outlook 2020. *CompTIA.* [Tinkle] 2020 m. Gruodis. https://www.comptia.org/content/research/it-industry-outlook-2020.

[DRF+06] *VIRTUAL PROTOTYPING FOR VEHICLE DYNAMIC MODELLING.* **Drivet, A., et al.** 2006 m., IFAC Proceedings Volumes, Volume 39, Issue 16, p. 986-991.

[DVB19] *Review of Research into the Nature of Engineering and Development Rework: Need for a Systems Engineering Framework for Enabling Rapid Prototyping and Rapid Fielding.* **Dullen, Shawn, Verma, Dinesh ir Blackburn, Mark.** 2019 m., Procedia Computer Science, Volume 153, p. 118-125.

[EA16] *Parametric optimization in virtual prototyping environment of the control device for a robotic system used in thin layers deposition.* **Enescu, Monica ir Alexandru, Catalin.** 2016 m., IOP Conf. Series: Materials Science and Engineering.

[Hei19] **Heintz, Meghan.** Launching a new warehouse with SimPy at Rent the Runway. s.l. : PyData New York City 2019, 2019 m.

[Yu20] *Towards fast prototyping of cloud-based environmental decision support systems for environmental scientists using R Shiny and Docker.* **Li, Yu.** 2020 m., Environmental Modelling & Software, Volume 132.

[KBK+17] *Towards a simulation-based framework for decision support in healthcare quality assessment.* **Kisliakovskii, Ilia, et al.** 2017 m., Procedia Computer Science, Volume 119, p. 207-214.

[Lau20] *IT Project Failures, Causes and Cures.* **Lauesen, Soren.** 2020 m., IEEE Access, vol. 8, p. 72059 - 72067.

[LRM+21] *Open-source discrete-event simulation software for applications in production and logistics: An alternative to commercial tools?* **Lang, Sebastian, et al.** 2021 m., Procedia Computer Science, Volume 180, p. 978-987.

[MDT22] **Matplotlib development team.** Matplotlib: Visualization with Python. *matplotlib.* [Tinkle] 2022 m. https://matplotlib.org/.

[MSM08] *Application of virtual prototyping for optimization of fuzzy-based active suspension system.* **Montazeri-Gh, M., Soleymanizadegan, Mehdi ir Mehrabi, Naser.** 2008 m., Proceeding of the 5th International Symposium on Mechatronics and its Applications, ISMA 2008, p. 1-6.

[Num22] **NumPy.** NumPy documentation. *NumPy.* [Tinkle] 2022 m. https://numpy.org/doc/stable/index.html.

[OGY+14] *Implementing ManPy, a Semantic-free Open-source Discrete Event Simulation Package, in a Job Shop.* **Oladipupo Olaitan, John Geraghty, et al.** 2014 m., Procedia CIRP, Volume 25, p. 253-260.

[PCB16] *Forest-based supply chain modelling using the SimPy simulation framework.* **Pinho, Tatiana M., Coelho, João Paulo ir Boaventura-Cunha, José.** 2016 m., IFAC-PapersOnLine, Volume 49, Issue 2, p. 90-95.

[Phi21] **Philips.** How Philips Hue works. *Philips.* [Tinkle] 2021 m. Birželis 12 d. https://www.philips-hue.com/en-us/explore-hue/how-it-works.

[RVS+18] *Rapid prototyping of distributed embedded systems as a part of Internet of Things.* **Raskin, Denis, et al.** 2018 m., Procedia Computer Science, Volume 135, p. 503-509.

[Sci22] **ScienceDirect.** Poisson Distribution. *ScienceDirect.* [Tinkle] 2022 m. https://www.sciencedirect.com/topics/mathematics/poisson-distribution.

[TSP20] **Team SimPy.** Simpy - Discrete event simulation for Python. *SimPy.* [Tinkle] 2020 m. Gruodžio 13 d. https://simpy.readthedocs.io/en/latest/.