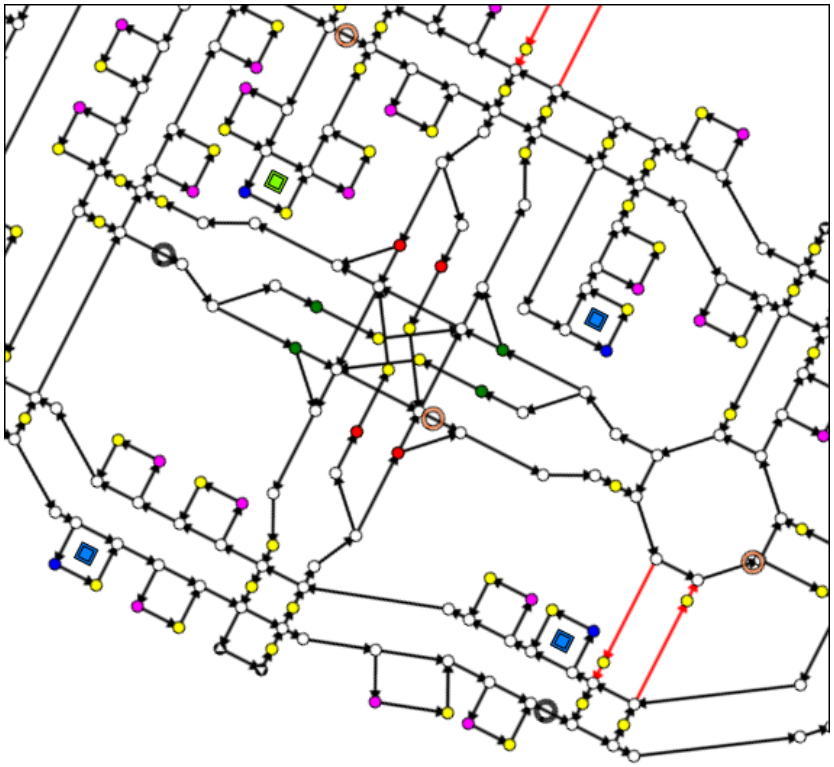
Vedligeholdelse af Trafik Simuleringsmodeller

Project Report
Group A319



Aalborg University
Det Teknisk-Naturvidenskabelige Fakultet
Strandvejen 12-14
DK-9000 Aalborg

Det Teknisk-Naturvidenskabelige
Fakultet AAU
Strandvejen 12-14
DK-9000 Aalborg
http://cs.aau.dk

AALBORG UNIVERSITY
STUDENT REPORT

Title:
Vedligeholdelse af Trafik Simuleringsmodeller

Theme:
Simulering

Project Period:
2. Semester 2016, P2

Project Group:
A319

Participant(s):
Benjamin Jhaf Madsen
Jacob Sloth Thomsen
Alexander Umnus
Kim Larsen
Lasse Fisker Olesen
Niclas Struntze Bach
Rasmus Thomsen

Supervisor(s):
Anders Mariegaard

Copies: 5

Page Numbers: 76

Date of Completion:
May 23, 2016

Abstract:

This report is written for the 2nd semester project of group A319. The theme of the work done and described within this report is "Simulering" and shows the extensive work there has been done on analyzing and idenfitfying a problem within the field of simulations as well as describe the development of a solution to said problem. The report is split into two parts, the problem analysis in which the process of going from an initial problem-statement to a descriptive final problem-statement is documented. The second part of the report describes the developed solution and the the group's decisions that were taken when making the solution. Requirements and success-criteria were made according to the findings made in the problem analysis and the final problem-statement. Furthermore, this part of the report compares two shortest-path algorithms the group looked at for implementation of the program, A* and Dijkstra, and also details the decision made on this matter. Most importantly, this part also includes detailed descriptions of some of key parts of the developed program, as well as snippets of the sourcecode for these parts.

Contents

1 Indledning

Trafiksystemet i Danmark undergår ofte udbygninger og ændringer, hvilket kan påvirke nærliggende vejnet. Det kan være svært at se hvordan disse ændringer vil påvirke trafikken, og derfor er der blevet opstillet forskellige modeller til at forudsige hvordan trafikken på vejnettet i fremtiden vil afvikle sig. Disse trafikmodeller bliver ofte opbygget med et konkret formål i fokus, og bliver derfor svære at vedligeholde i fremtiden i de tilfælde hvor konteksten ændrer sig. Konsekvensen af dette er at under en tredjedel af modellerne er blevet vedligeholdt, og at der ikke længere findes en model der dækker hele Danmark [20, s. 1-2].

1.1 Problemets Relevans

Formålet med trafikmodellerne er at forhindre trafikpropper og at sænke rejsetiderne. Uden modellerne er det besværligt at bestemme hvor der er problemer i vejnettet, og hvilken effekt nye veje vil have på trafikstrømmen. Trafikpropper har en effekt på landets økonomiske vækst, forurening, livskvalitet og tiden det tager for beredskaber og politi at nå frem.

Man kan risikere at sidde fast i trafikken på vej til arbejdet. For at finde ud af hvilken effekt denne spildtid har på Danmarks økonomi, har Michael Knørr Skov og Karsten Sten Pedersen, der arbejder for konsulent firmaet COWI, analyseret 3 vejprojekter [15]. Vejprojekterne inkludere en tredje Limfjordsforbindelse, en ny motorvejsstrækning ved København, og en Forbindelse mellem Fyn og Als. Udfra COWI's beregninger vil disse tilføjelser spare danskere 25 tusinde timer dagligt, hvilket svarer omtrent til en værdi på 2500 millioner kroner årligt. Antager man at en fjerde del af denne tid bliver brugt på arbejde vil man opleve en BNP-vækst på 0,035%. Et velfungerende vejnet er dermed et vigtigt aspekt i forhold til at forbedre Danmarks økonomiske vækst.

Billister er en af de største kilder til CO_2 forurening. Mængden af CO_2 der bliver udsluppet, afhænger af hastigheden billisterne kører. Ved en lav hastighed kan CO_2 udslippet per kilomet blive fordoblet, i forhold til at køre en stabil 50-

130 km/t. I den anden ende, hvis man kører over de 130 km/t vil udslippet igen øges, da bilen er mindre effektiv i udnyttelsen af brændstoffen [14, s. 5-6].

En undersøgelse har vist at der er en sammenhæng mellem trafik densiteten, og stress niveauet på en individ der befærder sig i denne trafik. Udover at det kan være ubehageligt under kørslen, bliver stressen også ført med videre på arbejdet og til hjemmet. Stressen kan også føre til aggresiv kørsel og i værste tilfælde ender det med en ulykke [13, s. 2-3].

1.2 Initierende problemformulering

Danske trafik modeller bliver ofte opbygget med et konkret formål i fokus og bliver svære at vedligeholde i fremtiden i de tilfælde hvor konteksten ændrer sig.

Arbejdsspørgsmål

- *Hvilke ændringer i konteksten skal der tages højde for ved trafik simulering*
- *Hvordan fungerer de forskellige eksisterende software, trafik modellerings simulatorer?*
- *Hvem har gavn af trafikmodellerings simulatorer?*

2 Simulering

Simuleringsmodellering og analyse er en del af processen, når man skal programmere en matematisk model af et fysisk system. Et system er defineret som en samling af dele/komponenter som modtager en form for input og så giver output. Dette output kan så bruges til forskellige formål. Man kan ved hjælp af data analysere de forskellige problemer i virkeligheden for at kunne foretage sig vigtige valg inden for drift- eller politiske ressource beslutninger. Man kan også benytte simulatorer til at træne folk i at tage bedre beslutninger eller forbedre ens egen ydeevne inden for et område. [3, s. 16-20]

En anden form for simulering er computer simulation. Simulation er også baseret på eksisterende eller foreslået system. I modsætning til simulationsmodeller, hvor valgene er lavet på forhånd, bliver valgene for en simulator genereret mens simulationen er i gang. Meningen med computer simulation er ikke at lave en beslutning, men at udsætte nogle personer for et system og træne deres evne til at lave gode beslutninger. Disse simulatorer er ofte kaldt træningssimulator. Meningen bag modellering og analyse af forskellige typer af systemer er:

- Få indblik i hvordan systemet fungerer
- Udvikling af drift og ressource beslutninger til at forbedre systemets ydeevne
- Afprøvning af nye koncepter eller systemer før implementering
- Opnå information uden at påvirke det rigtige system

Fordelene ved at benytte simulering kan være:

- Hurtig resultater fremfor i virkeligheden
- Det er lettere at analysere et system
- Nemt at demonstrere hvordan modellen fungere

Hurtigere resultater fremfor i virkeligheden Igennem simulering af et system kan man bestemme nogle variabler, det kan være f.eks. tiden som

simulationen skal vare over. Dette betyder at man hurtigere kan få resultater fra simuleringen, og så kan man lave flere gentagelser for at kunne komme frem til den bedste metode. Da før i tiden, var det ikke muligt at lave analyse på nogle systemer som varede over langt tid.

Det er lettere at analysere et system Før i tiden, hvor man ikke benyttede computeren til at lave en simulation over et system, så krævede det mange ressourcer at kunne analyser et problem. Hvis man skulle have en meget kompleks system analyseret, så skulle man kontakte matematikkere eller forsknings analytikere. Nu hvor man kan lave en simulering over computeren, så er der flere personer der kan få analyseret noget, da kravene er blevet reduceret.

Nemt at demonstrere hvordan modellen fungere De fleste simuleringssoftware har den egenskab, at kunne animere model grafisk. Amimationen er både brugbart for at kunne debugge modellen for at finde fejl, men også for at kunne demonstrere hvordan modellen fungerer. Amimationen kan også hjælpe med at se hvordan systemet opfører sig i løbet af processen. Uden muligheden for amimation, så ville simuleringsanalyse være sværere at forklare uden at bruge visuelle hjælpemidler.

Ulemperne ved at benytte simulering kan være:

- Simulering kan ikke give et præcist resultat, hvis input data ikke er præcise
- Simulation kan ikke give simple svar på komplekse problemer
- Simulation alene kan ikke løse problemer

Simulering kan ikke give et præcist resultat, hvis input data ikke er præcise Uanset hvor god modellen er, så er det ikke muligt at forvente et præcis svar, hvis man benytter upræcise svar. Man kan omskrive det til "Garbage in, garbage out". Desværre er opsamling af data set som den sværeste del i processen i at lave en simulering. Nogle som står bag simulering af et system har accepteret at nogle gange er man nødt til at bruge historisk data på trods af tvivlsom kvalitet for at spare dataindsamling tid. Hvilket kan føre hen til en mislykket simuleringsprojekt [3, s. 16-20].

Simulation kan ikke give simple svar på komplekse problemer Nogle analytikere kan tro på at simuleringsanalyse vil give give simple svar på komplekse problemer. Men det er et faktum, at når man har med komplekse problemer at gøre, så får man komplekse svar. Det er muligt at simplegøre resultaterne, men det kan risikere at undlade nogle parametere som gør projektet mindre effektiv.

Simulation alene kan ikke løse problemer Nogle ledere der står bag et projekt, kan tro at alene at gennemføre simuleringsmodel og analyseprojekt kan føre til at kunne løse et problem. Men simulering af et problem, kan føre til mulige løsninger til et problem. Det er op til dem der står bag problemet om de så vil benytte nogle af de løsninger de er kommet frem til. Ofte så de løsninger

man kommer frem til bliver aldrig eller dårligt brugt, på grund af organisatorisk passivitet eller politisk overvejelser. [3, s. 20]

Part I

Problemanalyse

3

Indledning til problemanalyse

I denne del af rapporten vil man komme i dybden i hvilke ændringer der sker i konteksten, der skal tages stilling til, når man skal simulere trafik. Der vil også være en analyse af to programmer som der bliver benyttet indenfor trafiksimulering. Senere kommer der en redegørelse om de forskellige interessenter, som enten kunne blive påvirket af vores program, eller påvirker vores program. Til sidst vil der være en endelig problemformulering, hvor man er kommet frem til specifikt område som rapporten vil fokusere på.

4

Ændring af Konteksten

Som der beskrives i den initierende problemformulering, er det vanskeligt at vedligeholde trafikmodeller, på grund de ændringer der sker i konteksten. Disse ændringer er fundet til at være en vækst i bestanden af køretøjer, og et skift i hvilke transportmidler der bliver benyttet af Danskere.

Data fra Danmarks Statistik viser en vækst i bestanden af køretøjer som ses på figur 4.1. I tilfælder hvor en trafikmodel ikke tager denne vækst med i overvejelserne, kan det føre til et urealistisk billede af virkeligheden, i det at der sandsynligvis vil være mindre stress på vejnettet.

Hvis man sammenligner antallet af køretøjer med befolkningsantallet, vil man se en stigning i køretøjer per borger. I 1995 ejet 41% af borgerne i Danmark et køretøj, og i 2016 er dette steget således at 55% af borgerne i Danmark ejer et køretøj. For trafikmodeller der kun undersøger afviklingen af biltrafik vil dette ikke have nogen påvirkning, men for modeller der inddrager andre transportmidler vil dette skift skulle tages med i beregningerne, hvis der bliver kigget på fremtiden.

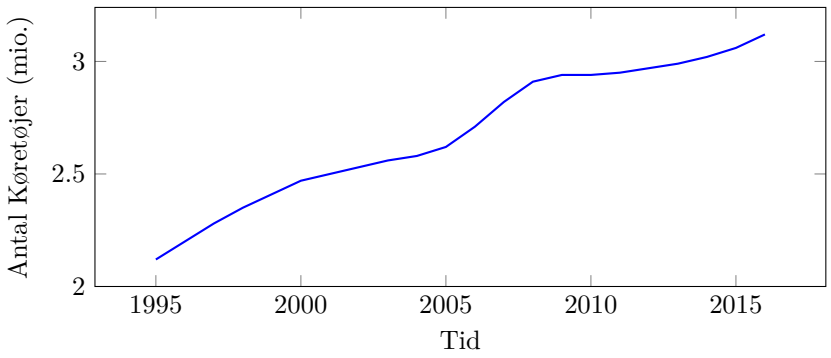


Figure 4.1: Væksten i bestanden af køretøjer (Statistikbanken)

4.1 Transportmiddelvalg

Det kan argumenteres, baseret på vedligeholdelsesparametre angivet heri at baggrunden for en trafikmodel der kan være fremtidigt brugbar afhænger af andre tilfælde end lige netop antallet af biler i forhold til befolkningen. Det kan også anskues at andre herunder givne parametre kunne gøre simulering af disse trafikmodeller en vanskelighed, hvori en drastisk ændring kunne medføre grunde til fejlkilder ved vedligeholdelse af de førhen nævnte modeller.

- Transportmidlers popularitet
- Ny teknologi
- Adfærd

Under disse parametre kan der uddrages en række kategorier af transportmidler der er gældende for trafikmodellerne, dette kan også forstås som alle nuværende relevante transportmidler for persontransport. Disse kan lægges under kategorierne kollektiv transport (busser, toge mm.) og personlig transport (personbiler, cykler mm.)

Transportmidlers popularitet kunne indebære en stigning i behovet for at benytte cykler eller lignende. Denne parameterændring ville være et essentiel eksempel at bearbejde til at videregive en bedre vedligeholdelsesstandard, antaget at der findes grundlag for dette.

Ny teknologi indebærer forbedringer til nuværende transportmuligheder og integration med nuværende orienteringsværktøjer som f.eks. applikationer. Simulering af denne parameterændring er uforudsigelig i bedste tilfælde, dog kan undersøgelser henligge til mulig nye teknologier der kunne have relevans.

Ud fra disse kategorier er der udvalgt en række transportmidler. Transportmidlerne er baseret på kategoriernes prioritet i forhold til relevans for løsningsmodellen. Kollektive og personlige transportmidler kan stilles op foran hinanden og argumenteres på baggrund af forudsigelige ændringer i parametre. Herunder grundlaget beliggende i væksten af priser mellem de forskellige transportmuligheder og parameterændringer for på et samfundsmæssigt og teknologisk grundlag.

4.2 Eksisterende Modeller

De modeller der er vedligeholdt og stadig bliver brugt i dag, er meget forskellige i deres fokus. Der findes modeller som Senex, der analysere godstrafikken mellem Danmark og Tyskland, der er en meget advanceret model til trafikafviklingen i hovedstadsområdet, og en masse mindre regionale og kommunale modeller [20, s. 2]. Forskellen på modellerne kan ses på detaljeringsgraden og hvor langt modellen kigger ud i fremtiden, hvor de mindre modeller har flere detaljer, men kun kigger få år ud i fremtiden, og vice versa for de større modeller. Trafikmodellerne er derfor delt op i 3 niveauer; makro-, meso- og mikroniveau.

Makroniveau På makroskopiske niveau er de anvendte modeller langsigtede, men med færre detaljer. Manglen på detaljer er påkrævet, da det ellers vil blive for svært at anskaffe data'en, der skal bruges til at specificere alle forudsætningerne for de anvendte modellers forudsigelser [20, s. 1] Modeller på makroniveau danner et billede over den internationale situation [20, s. 9]. En af de anvendte modelmetoder på dette niveau er prognosemodeller, disse benyttes til at beregne fremtidens trafik behov, således at der kan planlægges hvordan ressourcerne kan anvendes.[16]

Mesoniveau På det mesoskopiske niveau er detaljerings graden højere i forhold til makroniveau. De anvendte modeller på dette niveau er 4-trinsmodeller. Disse modeller bliver anvendt til at forudse trafikkens bevægelsesmønster.[16] De bruges også til at finde ud af hvilke veje der er belastede eller hvor lang tid en rejse vil tage. Disse modeller bliver brugt til at vise udviklingen i både internationale, nationale og regionale situationer [20, s. 9].

Mikroniveau På det mikroskopiske niveau er det kortsigtede modeller der bliver anvendt. Det område man undersøger er meget afgrænset, f.eks. vil man kun kigge på et enkelt lyskryds el. lign. Fordelen ved disse modeller er at den høje detaljerings grad kan give et mere præcist billede over situationen, dog kræver det at der skal bruges en masse data for at resultatet bliver realistisk [20, s. 9]. En af de anvendte moddelleringsmetoder er den empiriske metode, som bygger på observationer. Her er det enkeltstående situationer der bliver observeret, for at se hvilket udfald simuleringen har. På det mikroskopiske niveau kan det evt. være rundkørsler, lyskryds, eller en enkel vej der bliver analyseret.[16]

Gennem problemanalysen vil en af de vedligeholdte modeller og en af de ikke vedligeholdte modeller blive undersøgt ved hjælp af en teknologianalysen, 5, for at finde ud af hvilke elementer der er vigtige. Der er valgt at undersøge VisSim som en af de vedligeholdte programmer, og Altrans som en af de ikke vedligeholdte. VisSim er valgt, da det er et program der har eksisteret siden 1989 og stadig bliver anvendt idag. Altrans er valgt, da der var mange offentliggjorte informationer om programmet. Informationen fra teknologiananlysen kan derefter bruges til at lave en trafikmodel der kan vedligeholdes.

5 Teknologianalyse

I dette kapitel vil der blive gennemgået en analyse af simulationsprogrammerne VisSim og Altrans. En mangel på informationer om andre programmer har i en stor grad påvirket valget af VisSim og Altrans, derudover er de to programmer forskellige i deres formål, hvor Altrans har et specifikt formål, og VisSim er beregnet til at kunne løse flere forskellige formål. Altrans bliver ikke længere vedligeholdt, mens VisSim stadig bliver brugt i dag. Begge programmer vil blive vurderet i forhold til de aspekter der gør dem nemme at vedligeholde.

Inden da vil vi beskrive gruppens definition på hvad vedligeholdelse betyder for gruppen. Da der er tale om programmer og simuleringsmodeller, er der ikke tale om slid og skader, dog er der stadig tale om at sørge for at bevare disse løsninger. Derfor, når vi kigger på de to følgende to eksisterende løsninger, bliver de bedømt ikke kun på design, funktionalitet og deres kompleksitet, men også den eksisterende løsnings evne til at fortsat kunne blive anvendt ud i fremtiden.

5.1 VisSim

VisSim er et mikrosimuleringsprogram, som bliver anvendt i Danmark. VisSim udgør en stor del af beslutningsgrundlaget for udvidelsen i trafikken i dag. Programmet bruges til at konstruere og simulere større dynamiske systemer. VisSim er et diskret simuleringsprogram som modellerer adfærden for den enkle billist. VisSim benyttes for general modellering, simulation og designe simulations applikationer, dvs. at VisSim ikke nødvendigvis bruges til trafik simulering. VisSim er programmeret i ANSI C, og under processen af et VisSim projekt kan projektet kompileres.

VisSim benytter sig af psyko fysisk model, som benytter en regelbaseret algoritme ved bevægelser på tværs af banerne. Den psykologiske del bliver brugt til bilistens ønske om aggressivitet, hastighed, reaktionsevne og generelt menneskelige forhold til trafikken. Den fysiske del bruges til bilens adfærd, så som bilens hastighed, størrelse, position. [16]

Herunder på figur 5.1 kan det ses at VisSim består af en værktøjslinje, som

repræsentere kommandoer og blokke. Disse blokke og diagrammer bruges til at forme simuleringen. Det er et blokprogrammerings sprog, man programmere ved brug af blokke og diagrammer. På figur 5.1 kan man se at VisSim består af forskellige blokke, disse blokke er forskellige parametre og variabler, som udformer det kørende program [23].

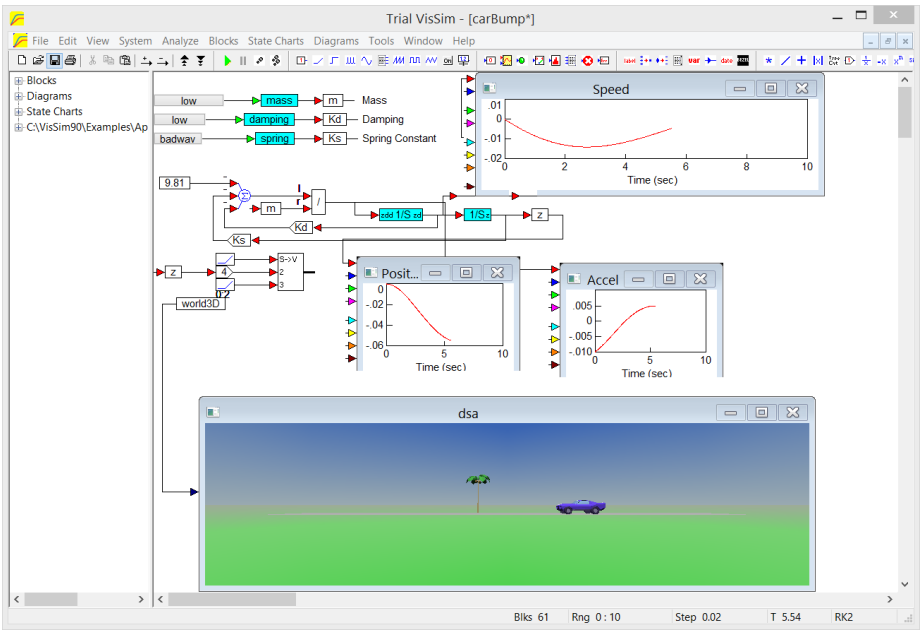


Figure 5.1: Værktøjer i VisSim

VisSim - Bilen

Den enkle bil spiller også en stor rolle i VisSim, bilen er bestående af forskellige parametre, og det er ofte disse parametre der bliver anvendt. Denne rapport vurdere nogle af VisSims parametre, da det ikke er muligt for os at vurdere alle parametrene, fordi rapporten også fokusere på andre aspekter end VisSim. Der vil også vurderes vedligeholdelsen af VisSim. Bilens parametre i VisSim er beskrevet nedenfor.[16]

- Ønsket acceleration.
- Deceleration.
- Acceleration
- Vægtfordeling.
- Hastighedsfordeling.
- Afstand mellem køretøjer.
- Størrelsen på køretøjet.

VisSim - Netværket

Netværket er bestående af de visuelle elementer, som har indflydelse på trafikafviklingen. Der er valgt at beskrive disse parametre, da det er disse der udgør største delen af VisSims parametre. Netværkets parametre er beskrevet nedenfor.

- Rundkørsler.
- Vigepligt.
- Lyskryds(signalregulering).
- Hastighedszone.
- Vejbredde.
- Vejlængde.

Analyse af acceleration og deceleration

Ud fra en undersøgelse foretaget af Pihlkjær afgangsprojekt Aalborg Universitet - Vej og Trafikteknik, viser det sig at nogle af VisSims accelerations og decelerations værdier kan være upræcise. Undersøgelsen er foretaget ved analysering af VisSim på de danske vej-netværk, hvor der sammenlignes med GPS acceleration og decleration med VisSims data. Her er der blevet indsat GPS i 166 bilister som skal repræsentere acceleration og deceleration i Danmark.

På figur 5.2, kan man se at accelerations fordelingen for VisSim er markant højere end GPS daten. Dette viser sig, at være pga. VisSim er henvendt til de tyske-vejnetværk. I undersøgelsen beskriver de, at det skyldes de tyske biler er større og hurtigere. Det vurderes at VisSims data er upræcis, dette kan resultere forkerte simuleringer, hvilket resultere i forkert planlægning af nye vejnetværk. [16]

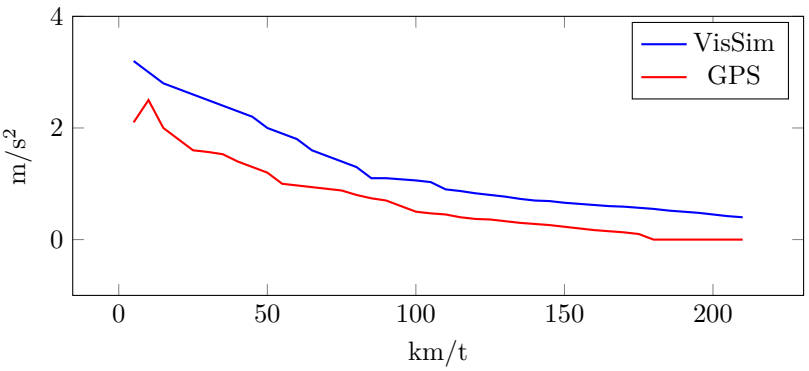


Figure 5.2

Samtidig er der undersøgt af Pihlkjær afgangsprojekt Aalborg Universitet - Vej og Trafikteknik om deceleration for bilister er præcise i VisSim, dette er også gjort ved sammenligning af VisSim data, med GPS data. På figur 5.3 kan man se, at VisSims deceleration er markant højere end GPS dataen, man ser også at VisSim er lineæret udspillet. Det vurderes at dette kan have betydning for

udfaldet i simleringen, hvis VisSims data var nær GPS daten, så ville udfaldet blive mere præcist. Man ser også at VisSims data er konstant, dette betyder at programmet ikke varierer decelerationen i forhold til farten. [16]

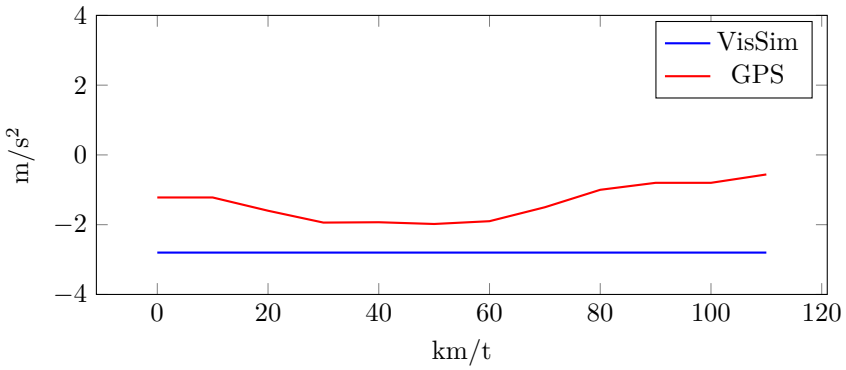


Figure 5.3

Vurdering

VisSim kan anvendes således, at det kan opstilles til hvilket som helst formål. Dette er en god detalje for VisSim, da brugeren selv kan opstille specifikke scenerier og simuleringer som ikke nødvendigvis er trafik relateret. Dette vurderes til at være godt for vedligeholdelsen, da brugeren selv har muligheden for at opstille scenarier. Dette gør VisSim fremtidssikret, således at udviklerne af VisSim ikke selv skal sørger for vedligeholdelsen.

VisSim er et avancerede program dette kræver viden omkring anvendelsen af VisSim for at brugeren kan benytte programmet til simulering af trafik. Da VisSim ikke kun er beregnet til trafik simulering, så kræver programmet mange detaljer og viden for brugeren, for at udføre en trafik simulering. Der vurderes at det vil være hjælpsomt for brugeren, hvis brugeren ikke skal anvende tid på at lære et nyt program, men i stedet anvende et program, som kun er egnet specifikt til simulering af trafik.

Vi har opstillet en tabel over de overvejelser vi har gjort os omkring VisSims fordele og ulemper.

Fordele	Ulemper
Kan tilpasses til en givet kontekst	Besværligt at sætte sig ind i
Brugeren kan programmere logik	Ikke brugervenligt
Vedligeholdes af brugeren	
Kan give meget præcis data	

Table 5.1: VisSim Fordele og Ulemper

5.2 Altrans

Alternativ Transportsystemer (Altrans) er en trafikmodel, hvis formål er at belyse hvordan en øget brug af den kollektive transport vil påvirke miljøet [10, s. 14]. Trafikmodellen er blevet udviklet af Danmarks Miljøundersøgelser (DMU) i 1994. Altrans består af 3 hovedmodeller: en geografisk model, en adfærds model og en emissions model [9, s. 14]. Emissions modellen bliver ikke nævnt i rapporten, da den ikke er relevant for dette projekt.

Geografisk model

Den geografiske model bruges til at beregne rejsetider, ventetider og skiftetider. For at udregne disse benyttes følgende undermodeller:

- Model af kollektiv transportnet
- Model af serviceniveau
- Model for bilrejser
- Model for attraktion til byfunktioner

Modellen for attraktion til byfunktioner består af information over antallet af beboere og arbejdspladser i forskellige områder, og benyttes mest i forbindelse med adfærds modellen. Derudover bruger den geografiske model et geografisk informations system (GIS) til opbevaring af data og udregning af rejsetiderne [10, s. 18-19].

For at lave realistiske simuleringer, bruger modellen for det kollektive trafik præcise data for ankomst- og afgangstider. Denne data kommer fra 11 trafikselskaber der bruger køreplansystemet TR-System, DSB's data kommer i et andet format der bliver brugt til DSB's egen rejseplanlægger. For at beregne rejsertiderne af kørestrækningerne, samt ankomst og afgangstiderne sat op i et tredimensional koordinatsystem, hvor tiden bliver indsat som den tredje akse Z. Stationernes placering bliver indsat som X og Y koordinaterne, og man kan dermed finde ud af hvilke ruter der kan rejses med ved en given station [10, s. 20-22].

Modellen af serviceniveau udregner serviceniveauet med variablerne tid, omkostninger og tilgængelighed. Modellen undlader at inkluderer variabler som komfort, da DMU har lavet antagelsen, at komforten ikke ændre sig kraftigt over tid. Dette kan gøre prognoser der ser på den fjerne fremtid upræcise. Modellen spænder over både taktisk (meso) og operationel (mikro). På den taktiske plan kigger Altrans på buskilometer, afgangs frekvenser og tilgængelighed. På den operationelle plan kigges der på tiden man bruger i køretøjet, hvor lang tid man skal vente ved skift samt ventetiden i alt, og prisen på rejsen [9, s. 36-37].

Formålet med modellen for bilrejser er at udregne tiden det tager at rejse fra by til by. Dette gøres ved brug af vejnettet i GIS, og hastigheden bilen kører kommer an på vejtypen. Ruten der bliver kørt starter og slutter fra centrumet af byerne der bliver rejst mellem. Modellen tager ikke højde for anden trafik på

vejene, så tiderne der udregnes vil være præcise hvis der ikke er andre bliver på vejnettet [10, s. 25]. Hastighederne modellen bruger til de forskellige veje, kan ses på tabel 5.2.

Motorveje	110
Motortrafikveje	90
Hovedveje	80
Øvrige veje på landet	70
Veje i byer	40

Table 5.2: Hastigheder på forskellige vejtyper

Adfærds model

Adfærds modellens formål er at give et estimat på fordelingen af transportmiddelvalg, populariteten af destinationer, kørekort fordeling, og bilejerskab. For at udregne estimaterne, benytter adfærds modellen sig af 3 undermodller [10, s. 25-26]:

- Model for valg af transportmiddel og destinationer
- Cohortmodel og model for kørekorthold
- Model for bilejerskab

Modellen for valg af transportmiddel og destinationer estimerer og simulerer antallet af kilometer der bliver rejst i de 4 transportmiddelkategorier: kollektiv trafik, bilfører, bilpassager og let trafik. Derudover estimeres de rejsenes destinationer, hvilket gør det muligt at finde ud af hvordan trafikken bliver fordelt på vejnettet, så trafikkens påvirkning på miljøet kan analyseres. Hovedformålet med at finde destinationerne er dog at modellere indvider i samfundet. Modellen vægter nytten ved rejserne, for eksempel kan en rejse til den nærmest købmand være mere nyttig end en der ligger længere væk. For at finde ud af hvilket transportmiddel et individ vælger, kigges der på prisen og tiden af rejsen, samt individets socioøkonomiske baggrund [10, s. 26-27].

Modellen for kørekorthold er en prognosemodel. Sandsynligheden for at et individ har et kørekort, er udregnet ud fra kørekortfordelingen over alle individer og en logitmodel med variablerne køn, alder, indkomst, stilling og urbaniseringsgrad. Dette inddrages i cohortmodellen der simulerer om individet har et kørekort i det år der bliver beregnet på [10, s. 30].

Modellen for bilejerskab estimerer hvor mange biler en husstand har. Modellen består af en logitmodel der bestemmer hvor mange biler husstanden har, denne logitmodel er indlejret i anden logitmodel, der bestemmer hvorvidt husstanden har biler eller ej. Til at bestemme om husstanden har bil, kigges der på husstandens socioøkonomiske forhold, om individerne i husstanden har kørekort, og hvor individerne rejser til. Outputtet af denne model bruges efterfølgende I modellen for valg af transportmiddel [10, s. 29-30].

Vurdering

Når der i den Geografiske model bliver udregnet rejsetider for biler, bliver der ikke overvejet hvordan trafikken er på vejene. Rejsetiderne bliver udregnet ved at finde ruten gennem vejnettet og derefter gange delafstandene med hastighederne på figur 5.2. At udelade trafikdensiteten i udregningen kan dermed gøre bilrejser mere attraktive når et individ skal vælge transportmiddel, hvilket kan føre til et upræcist resultat. Ved simulering kan man selvfølgelig ikke lave en model der passer 100% på virkeligheden, men i dette tilfælde kunne modellerne muligvis have taget et andet advanceret simulerings program som for eksempel VisSim i brug til at udregne realistiske rejsetider. Et andet problem når der skal udregnes bilrejser, er at afstanden bliver udregnet fra centrum til centrum af byer. Dette kan give et urealistisk billede hvis individet egentlig kun skal fra udkanten af en zone til udkanten af en sidelæggende zone, specielt hvis rejsen foregår i kun et centrum, da modellen da vil tage gennemsnittet for rejser i det centrum. Derudover gør det, at individet bare skal køre til et centrum, at modellen ikke overvejer hvor langt individet skal gå fra en parkeringsplads til destinationen.

Adfærds modellen finder destinationen et individ rejser til, og hvilket transportmiddel der bliver valgt, men der bliver ikke overvejet om individet vil rejse eller ej. Det vil sige at alle individerne i simuleringen rejser på en beregnings tidspunktet. Dette kan gøre at både vejnettet og den kollektive trafik virker til at være mere belastet end de i virkeligeheden vil være. I forhold til Altrans formål, at finde miljøpåvirkningen i skift fra bilrejser til kollektiv transport, vil dette ikke gøre at resultatet bliver upræcist, hvis man kigger på dataene procentvis, men det vil være svært at bruge Altrans resultater i sammenspil med andre simuleringsmodeller, der overvejer hvor mange individer der rejser på en dag.

I forhold til vedligeholdelse, har Altrans følgende ulemper. Vejnettet og destinationerne bliver indlæst i et GIS system fra Transportvaneundersøgelsens data, hvilket betyder at det ikke er muligt selv at styre hvordan vejnettet ser ud. Dette kan blive problematisk da disse undersøgelser bliver foretaget med et 3 års interval, og man kan dermed risikere at arbejde med forældet data. Derudover er det er ikke muligt at specificere væksten af antallet af biler på vejnettet, da antallet afhænger af adfærdsmodellen. Det at man ikke kan definere denne vækst, kan være en af grundene til at modellen ikke længere bliver vedligeholdt, da den i fremtiden vil blive mere og mere upræcis. Generelt er Altrans meget fokuseret på hovedformålet, at finde ud af udviklingen i fordelingen af individer mellem den kollektive trafik og bilrejser. Hvis man skulle få brug for at vide hvordan denne fordeling påvirker vejnettet, kan man blive nødt til at bruge et andet simuleringsværktøj. Havde Altrans været mere fleksibel og spillet bedre sammen med andre simulerings modeller, vil der sandsynligvis være en større interesse i at vedligeholde den.

6 Interessantanalyse

I følgende afsnit vil der redegøres for hvem vi mener har en interesse i, at der bliver udviklet en softwareløsning som kan simulere forskellige instanser af trafikhændelser og hvordan nogle kunne se en interesse i at sågar modarbejde sådan et produkt. Dette er essentielt til at kunne opstille krav for sådan en løsning da det vil have konsekvenser for udviklingen af løsningen. Interessenterne beskrevet i følgende afsnit vil altså blive afgrænset således at softwareløsningen er passet til denne bestemte målgruppe om man vil.

6.1 Vejdirektoratet

Transport- og Bygningsministeriet(TRM) er Danmarks øverste danske statslige myndighed på transportområdet og bygningsområdet. TRM's hovedopgave er at sikre sig at de forskellige love bliver overholdt, ved opførelse af fx. en motorvej. Da TRM er en sammensætning af mange underdelinger har vi valgt at fokusere på en af deres styrelser nemlig Vejdirektoratet.

Vejdirektoratet står bag statsvejnettet som primært består af motorveje, hovedlandeveje og mange af landets broer. Alt i alt dækker disse forskellige veje 3.801 km vej [22]. Dette udgør i alt 5% af det offentlige vejnet men på trods af dette så er disse veje samtidig de veje hvor godt 50% af alt danmarks trafik forgår på. Vi mener at Vejdirektoratet er en væsentlig interessant netop da de er ansvarlige for planlægning af vejnettet i Danmark. Da vi agter at skabe en løsning der har funktionaliteten til at planlægge disse veje og skabe forskellige trafik scenarier til at simulere potentielle alternativer til at opbygge sådan en vejnet.

Med en interessant som Vejdirektoratet skal kvaliteten af softwareløsningen møde en hvis standard da løsningen gerne skulle konkurrere med allerede eksisterende værktøjer der anvendes af Vejdirektoratet.

6.2 Den Kommunale Sektor

Kommunerne er ansvarlige for at vedligeholde og oprette veje i de dele af Danmark der nu er afsat til dem. Kommunerne skal godkende oprettelse af nye veje i deres områder hvilket vil sige at der oftest er andre organisationer indblandet så som det førnævnte TRM. Planerne for disse veje er altså nogle som skal pitches til kommunen således man kan præsentere ens case for at der netop er brug for oprettelse af en vej. Til netop dette kunne kommunen have en interesse i at sådan en case bliver opstillet i et simuleringsprogram som vores hvori det vil fremgå hvordan ændringer/oprettelse af en vej ville udspille sig i teori. Det bliver nævnt i rapporten fra Danmarks TransportForskning at nogle enkelte af kommunernes modeller er blevet opdateret og vedligeholdt, f.eks. Odense er blevet opdateret i 2004. [20]

Dette betyder at vi anser den kommunale sektor som en mulig interessant i den kontekst at informationen fra vores løsning ville kunne argumenterer for en case om at nye veje skal oprettes. Dette betyder dog at den udviklede løsning skal kunne opnå en kvalitet hvori det bliver en anerkendt standard for præsenterbare, faktuelle simuleringer.

Visual Solutions

Visual solution er et blandt andre firmaer der har lavet forskellige simulerings værktøjer. VisSim er f.eks. et anerkendt værktøj som er anvendt af over 100.000 forskellige forskere verden rundt[7]. Vores simulerings program kan, med mere tid og udvikling potentielt blive en konkurrent til disse programmer og derfor menes det at dette firma er en modvirkende Interessant der kunne have en interesse i at modarbejde ideen.

Visual Solutions værktøj, VisSim er derfor bl.a. Også blevet analyseret i denne rapports Teknologianalyse. Som interessant kunne Visual Solutions prøve at modvirke løsningen udarbejdet som en del af dette projekt, potentielt kunne denne løsning blive en mulig konkurrent til VisSim hvilket potentielt kunne lede til at firmaet, alt efter teknologien udarbejdet på længere sigt, til at søge om at tilegne softwareløsningen.

6.3 Uddannelsessektoren

En løsning som den vi agter at lave i dette projekt kan også være et godt værktøj til uddannelse af folk der vil arbejde inde for trafik sektoren. Dette kunne bl.a. Være DTU Transport som forsker inde for transportområdet. DTU (Danmarks Tekniske Universitet) har før i tiden foretaget undersøgelse i sammenhæng med optimering af trængsel i trafik, miljøproblematikken og trafiksikkerhed[8]. I dette tilfælde ville værktøjet pivot mod en uddannelses kontekst hvilket på samme tid også kunne være en potentielt ide til videreudvikling. DTU er som sagt også ansvarlig for mange undersøgelse med anledning i trafik og kunne potentielt bidrage til udviklingen af softwareløsningen eller fremtidige iterationer af den. Studerende der læser til vej- og trafik teknik, kunne have interesse i at få et simuleringsværktøj, som kræver mindre viden i at bruge.

Uddannelsessektoren tilbyder en interessant mulighed til at udarbejde projektet i en anden retning.

6.4 Specificering af målgruppe

Ud fra interresantanalysen og teknologianalysen kan der delkonkluderes hvilken interresant der har størst interesse for vores projekt. Denne interresant er vurderet til at være kommunen baseret på at kommunerne står for størstedelene af vejnetværket i Danmark [2]. Derudover er den kommunale sektor anset for at være den mest realistiske interessant for gruppen, eftersom det er et skala vi har mulighed for at arbejde på. Det er blevet bestemt at et simulerings værktøj med netop dette fokus, ville gavne mest af at simulere i en mesoskopiske kontekst. Dette er yderlige uddybet i krav og specifikationer afsnittet i denne rapport. En god grund til at vi har valgt kommunerne som målgruppe, er at mange af kommunernes modeller ikke er blevet opdateret 6.2.

7 Problemformulering

Nuværende simuleringsværktøjer til simulering af trafik er enten sværere for nye brugere at anvende eller mangler fleksibiliteten til at kunne tilpasse sig den kontekst brugeren ønsker at arbejde i. Hvordan kan et mesosimuleringsværktøj, hvori brugeren gennem en brugerflade kan opstille et vejnet, samt indstille variabler som eksempelvis antallet af biler, hastigheder og adfærd, optimeres i forhold til vedligeholdese, trods ændringerne i konteksten?

Part II

Problemløsning

8 Teori

Dette kapitel beskriver teorien der er brugt til at udforme løsningen. Vi har i løsning brug grafteori til opstillingen af knuderne og vejene der der forbinder knuderne, samt brugen af grafen i A* til at finde den optimale rute gennem grafen. Psuedokoden vist slutning af afsnit 8.2 og 8.3, er skrevet med en stil der ligger op til objekt orienteret programmering. Her bruges `end` til at afslutte løkke og procedurer, og kommandoerne en if-sætning indeholder vises med indrykning.

8.1 Grafteori

Grafteori handler om at beskrive modeller matematisk. Grafteori er væslig når man skal optimere et netværk, det kunne f.eks. være en graf som representere et vejnetværk, hvor man her vil kunne bruge grafteori til bla rutefinding.

En graf beskriver et par af mægnder, man kunne tag udgangspunkt i grafen G = (V,E) hvor V og E er mægnder. I eksempet her er V en ikke-tom mængde af knuder, og E er mængden af kanter som forbinder knuderne i mægnden V.

Vi tager udgangpunkt i grafen på figur 8.1.

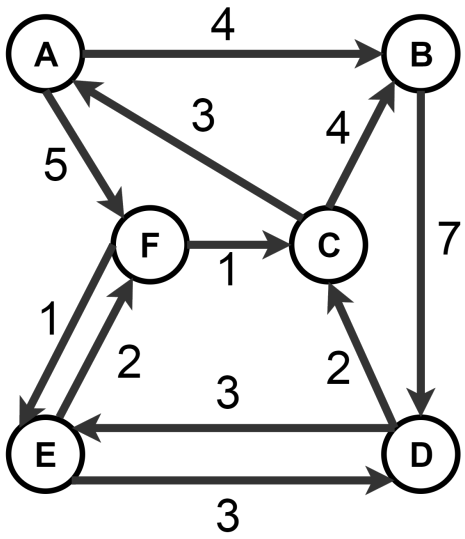


Figure 8.1: Orienteret vægtet graf

Man kan forstille sig dette er et vejnetværk, hvor knuderne repræsentere sving, og linjerne er veje som forbinder svingene, samt beskriver tallet mellem 2 knuder længden af vejen. Herved kan grafen godt forstille et simpel vejnetværk.

Vi kan ud fra grafen se at knuderne:

$$A, B, C, D, E, F \in V \tag{8.1}$$

Samt at kanterne

$$\{A, B\} \to 4, \{A, F\} \to 5, \{B, D\} \to 7, \{C, A\} \to 3, \{C, B\} \to 4, \{D, C\} \to 2, \\ \{D, E\} \to 3, \{E, D\} \to 3, \{E, F\} \to 2, \{F, C\} \to 1, \{F, E\} \to 1 \in E. \tag{8.2}$$

Grafen her kan derfor skrives rent matematisk:

$$G = (V, E), V = A, B, C, D, E, F \tag{8.3}$$

$$E = \{\{A,B\} \to 4, \{A,F\} \to 5, \{B,D\} \to 7, \{C,A\} \to 3, \{C,B\} \to 4, \{D,C\} \to 2, \{D,E\} \to 3, \{E,D\} \to 3, \{E,F\} \to 2, \{F,C\} \to 1, \{F,E\} \to 1 \in E\} \tag{8.4}$$

En anden måde at repræsentere grafen på er ved hjælp af en nabo-matrix VxE, hvor v1, v2...vn, er knuderne, og e1, e2...en, er kanterne. Se figur 8.2.

I matrixen beskrives forbindelser mellem 2 knuder med et tal, og knudere uden forbindelse beskrives med ∞. Man vil altså derfor kunne tegne en graf alene ud fra matrixens værdier.

$$G = \begin{array}{c} \\ A \\ B \\ C \\ D \\ E \\ F \end{array} \begin{array}{c} \begin{array}{cccccc} A & B & C & D & E & F \end{array} \\ \begin{bmatrix} \infty & 4 & \infty & \infty & \infty & 5 \\ \infty & \infty & \infty & 7 & \infty & \infty \\ 3 & 4 & \infty & \infty & \infty & \infty \\ \infty & \infty & 2 & \infty & 3 & \infty \\ \infty & \infty & \infty & 3 & \infty & 2 \\ \infty & \infty & 1 & \infty & 1 & \infty \end{bmatrix} \end{array}$$

Figure 8.2: Nabo Matrix af orienteret vægtet graf

Grafteori er et vigtigt emne, da f.eks. korteste rute algoritmer som Dijkstra's har brug for at vide hvordan knuderne er forbundet, samt graden(hvor mange kanter knuden har) af den enktle knude, for at kun udregne den korteste rute. Derudover vil en grafmetode som en nabo-matix være en god måde at beskrive et vejnætværk på programerings niveau, da man kan have alle sine værdier i en variable. [4] [5]

8.2 Dijkstras Algoritme

Dijkstra's er en grådig algoritme der kan finde en rute imellem to knuder i en vægtet graf. Grafen ruten kan findes i kan både være orienteret og ikke orienteret. Algoritmen starter med at sætte afstanden til alle punkter lig uendelig, udover start punktet da det er det eneste der kendes til på tidspunktet. Når algoritmen kører en graf igennem, kigger den på den knude hvor kosten er lavest, og udregner kosten til de naboliggende knuder. Når kosten udregnes for en nabo knude, tages kosten af ruten til den nuværende knude og afstanden til nabo knuden bliver adderet. Da algoritmen altid tager knuden med den lavest kost, vil den have fundet den korteste rute, når den nuværende knude er slutknuden [17, s. 681-684]. Som et eksempel kan vejen fra A til Z på figur 8.3 findes:

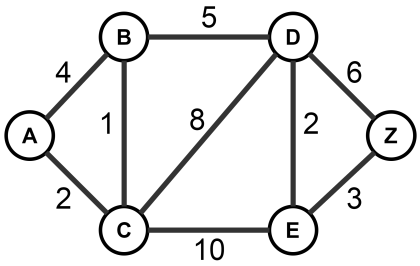


Figure 8.3: En vægtet ikke-orienteret graf

1. Algoritmen starter ved A, og kigger på naboerne B og C. Deres kost bliver noteret til at være 4 og 2.
2. Nu kigges der på knuden C, da det er den nuværende korteste rute. Her ses det at kosten til B er 3 (2+1), og det bliver overskrevet da det er mindre end den tidligere værdi 4. Deruodver noteres det at kosten for D og E er 10 og 12.
3. Så kigges der på knuden B, hvor det ses at kosten til D er 8 (2+1+5), og den overskriver den tidligere højere kost.
4. Derefter bliver knuden D kigget på, hvor den noterer kosten for E og Z til at være 10 og 14. Selvom der er fundet en vej til slutknuden, forstættes der, da kosten til knuden E er mindre end kosten til Z.
5. Så kigges der på noden E, hvor algoritmen ser at kosten til Z er 13, og dermed bliver den tidligere kost 14 overskrevet.
6. Der er ikke flere knuden der har en lavere kost en slutknuden Z, og den korteste rute er fundet.

Kostene for hver knude kan gemmes i en tabel, som set på tabel 8.1.

	A	B	C	D	E	Z
	0	∞	∞	∞	∞	∞
A	0	4	2	∞	∞	∞
A, C	0	3	2	10	12	∞
A, C, B	0	3	2	8	12	∞
A, C, B, D	0	3	2	8	10	14
A, C, B, D, E	0	3	2	8	10	13
A, C, B, D, E, Z	0	3	2	8	10	13

Table 8.1: Dijkstra Kost Tabel

```
1  object Node
2    Previous = null
3    Cost = PositiveInfinity
4    List NeighborNodes
5
6  procedure FindPath(AllNodes, StartNode, EndNode)
7    StartNode.Cost = 0
8    List Queue.Add(StartNode)
9      while Queue > 0 do
10       CurrentNode = Node with smallest Cost in Queue
11       if CurrentNode is EndNode
12         return TracePath(CurrentNode)
13       else
14         Queue.Remove(CurrentNode)
15         EvaluateNeighbors(CurrentNode)
16     end
17     return null // No path found
18 end
```

Figure 8.4: Dijkstra pseudo-kode: FindPath og Node

På figur 8.4 ses pseudokode for kernen af Dijkstras algoritme. En `Node` er defineret først, hvor `Previous` til at starte med er `null` eller 'ikke eksisterende', og `Cost` er uendelig. `FindPath` proceduren tager en graph `AllNodes`, startknuden `StartNode` og slutknuden `EndNode`. Listen `Queue` laves og `StartNode` indsættes, da den skal findes i første iteration af while løkken. While løkken køres mens der stadig er knuder der ikke er undersøgt, altså knuder der ligger i `Queue`, og den starter med at sætte `CurrentNode` til knuden der har den laveste `Cost`. Hvis `CurrentNode` er lig `EndNode` så har algoritmen fundet ruten og den bliver returneret ved brug af proceduren `TracePath`, som beskrives sidst i dette afsnit. Hvis ikke `CurrentNode` er lig `EndNode`, så fjernes knuden fra `Queue` listen og naboerne til knuden bliver evalueret gennem proceduren `EvaluateNeighbors` som beskrives herunder.

```
1  procedure EvaluateNeighbors(CurrentNode)
2    for each NeighborNode in CurrentNode.NeighborNodes
3      if NeighborNode is not in Queue
4        Queue.Add(NeighborNode)
5        PossibleCost = CurrentNode.Cost
6                       + distance from CurrentNode
7                         to NeighborNode
8        if NeighborNode.Cost > PossibleCost
9          NeighborNode.Cost = PossibleCost
10         NeighborNode.Previous = CurrentNode
11   end
12 end
```

Figure 8.5: Dijkstra pseudo-kode: EvaluateNeighbors

`EvaluateNeighbors` proceduren som set på figur 8.5, evaluerer naboerne af en `Node`, og hvis naboen ikker er i `Queue`, vil den blive tilføjet til listen. Mens evalueringen er igang så overskriver `Cost` og `Previous`, hvis `Cost` er lavere fra den nuværende knude `CurrentNode`.

```
1 procedure TracePath(CurrentNode)
2   List Path
3   while CurrentNode is not null
4     Path.Add(CurrentNode)
5     CurrentNode = CurrentNode.Previous
6   end
7   return Reverse(Path)
8 end
```

Figure 8.6: Dijkstra pseudo-kode: TracePath

Figur 8.6 viser proceduren `TracePath`, der finder vejen tilbage fra slutknuden. Dette gøres gennem en while løkke, der køres indtil at den forrige `Node` er lig `null`, hvilket vil sige at startknuden er nået. Knuderne overføres til en liste, der bliver invereteret og returneret til brugeren af proceduren.

8.3 A* Algoritmen

A* (A stjerne) er en udvidelse af Dijkstras algoritme. Forskellen ved A* algoritmen er at den estimerer hvor langt knuderne i graphen er fra slutknuden, og dermed findes den optimale rute hurtigere da den kun kigger på knuder der ligger i retningen af slutknuden. Dette gøres når nabo knuderne skal undersøges, ligesom i Dijkstra, vil algoritmen udregne kosten til nabo knuderne af det nuværende punkt, og derudover vil der benyttes en heuristik til at estimere kosten fra nabo knuden til slutknuden. Det vil sige at algoritmen arbejder med kosten til en knude, betegnet `G`, en heuristik der estimerer kosten fra knuden til slutknuden, betegnet `H`, og den samlede vurdering `F`, der udregnes som vist på formlen 8.5

$$F(n) = G(n) + H(n) \tag{8.5}$$

Hvis man ønsker at A* skal finde den absolut korteste rute, kræver det at heuristikken er optimistisk, altså at den aldrig overestimerer kosten til slutpunktet. Som et eksempel på en heuristik der kunne man definere heuristikken som værende afstanden i en lige linje til slutpunktet, da der aldrig ville være en kortere vej end dette. Yderligere, hvis man har informationen, kan der inddrages hastighedsgrænsen på vejene ved at dividere afstanden i en lige linje med den højste hastighedsgrænse der findes på vejnettet.

Estimationerne der bliver udregnet benyttes hver gang algoritmen skal vælge den næste `Node` der skal undersøges. Ligesom Dijkstra tager den knude der har den nuværende laveste `Cost`, tager A* den `Node` der har den laveste `Estimation F`.

```
1  object Node
2    Previous = null
3    Cost = PositiveInfinity
4    Estimate = PositiveInfinity
5    List NeighborNodes
```

Figure 8.7: A* pseudo-kode: Node

Forskellen på pseudokoden fra Dijkstras algoritme er lille. Til `Node` objektet, der ses på figur 8.7, er der tilføjet `Estimate` variablen, der senere bliver udfyldt af `EvaluateNeighbors`.

```
1  CurrentNode = Node with smallest Estimate in Queue
```

Figure 8.8: A* pseudo-kode: Find smallest

I stedet for at sortere køen på `Cost` som der gøres i Dijkstra, bliver køen nu sorteret på `Estimate`, som set på figur 8.8.

```
1  procedure EvaluateNeighbors(CurrentNode)
2    for each NeighborNode in CurrentNode.NeighborNodes
3      if NeighborNode is not in Queue
4        Queue.Add(NeighborNode)
5        PossibleCost = CurrentNode.Cost
6                       + distance from CurrentNode
7                         to NeighborNode
8        if NeighborNode.Cost > PossibleCost
9          NeighborNode.Cost = PossibleCost
10         NeighborNode.Previous = CurrentNode
11         NeighborNode.Estimate = NeighborNode.Cost
12                               + Heuristic(NeighborNode)
13   end
14 end
```

Figure 8.9: A* pseudo-kode: EvaluateNeighbors

Den sidste ændring er fortaget i `EvaluateNeighbors`, set på figur 8.9. Efter at en nabo `Node` er blevet fundet til at have en lavere `Cost` end den tidligere havde, bliver `Estimate` udregnet ved brug af knudens `Cost` eller `G`, plus heuristikken som eksempelvis kunne være distancen fra knuden til slutpunktet, som ses på linje 11 og 12 på figur 8.9.

8.4 Testing

Unit Test
En unit er det minste testdel af et program, altså det kan være funktioner, klasser, procedure eller interfaces. Unit testing er en metode man kan benytte når man vil teste hver inviduel del af programmet virker og om de er egnet til brug. Unit tests er skrevet og benyttet af programmører, for at være sikker på at ens kode opfylder de krav som er forventet af det.

Formålet med denne test er at splitte programmet op i mindre dele, hvor derefter man tester en efter en at de forskellige dele fungere optimalt og som de skal. Det kan være en funktion, som man vil teste, hvor man har nogle input og derefter skulle funktionen have de rigtige output fra funktionen. På den måde kan man håndtere fejl når det forkerte input er givet.

Fordelene ved unit testing:

- Problemer/fejl findes tidligt, så det ikke påvirker senere kode.
- Unit testing hjælper med at vedligeholde og nemt kan ændre koden.
- Opdagelse af bugs tidligt, hvilket hjælper med at reducere omkostningerne når man skal fejlrette koden.
- Unit test hjælper med at forenkle debugging processen, så hvis der sker en test fejl, så er det kun de seneste rettelser der skal rettes.

[18]

Gruppen vurderede andre test metoder igennem såsom Blackbox testing metoden. Grunden til at vi ikke benyttede os af denne metode er, fordi den kræver at vi har en person som skal teste programmets brugergrænseflade og hvordan programmet fungerer. Den person kender ikke til kodens struktur og hvordan det er blevet programmeret. Vi i gruppen er kommet frem til at Unit testen er en bedre og mere effektiv løsning til at teste vores program igennem. Da vores program benytter sig meget af forskellige klasser, metoder og interfaces, så mente vi som gruppen at den rette fremgangsmåde var at teste de forskellige dele i vores program del efter del. Da vi kan give input til programmet, og så finde ud af om det rigtige output kommer ud.

9 Design af program

Dette afsnit har til formål at udvikle et program, som imødekommer kravet om vedligeholdelse. Afsnittet vil sætte nogle kravspecifikationer og succeskriterier. Kravspecifikationerne er dem som skal forme programmet og samtidig gøre programmet realistisk. Formålet med programmet skal være at brugeren selv skal kunne indstille det efter sine behov. Programmet skal sættes op således det er TrafikTeknikere der kan anvende det og sammenligne to simuleringer, for at finde den mest optimale løsning til et vejnetværk i den realistiske verden.

9.1 Kravspecifikationer

I dette afsnit vil der blive forklaret hvilke kravspecifikationer programmet skal opfylde. På tabel 9.2 vises en samlet tabel over de kravspecifikationer der er sat for programmet. Tabellen er opdelte i 3 katagorier `Generelt`, `Brugerflade` og `Simulering`. I de følgende afsnit bliver tabel 9.2 beskrevet.

Generelt	**Brugerflade**	**Simulering**
Gem og Åben fil	Gitter	Sammenligning
	Noder	Fodgængere
	Veje	Cyklister
	Eksklusive veje	Køretøjer
	LysKryds	Pathfinding
	Vigepligt	Acceleration
	Huse	Deceleration
	Destinationer	
	Parkeringspladser	

Table 9.2: Kravspecifikationer

Generelt

Der er valgt at programmet skal indeholde en gem og åben funktion, dette skal gøres for at brugeren får muligheden for at gemme sit projekt. På denne måde

kan brugeren arbejde videre på sit projekt over en større tidsperiode. Der er også valgt at opstille dette, således brugeren har forminsket tidspild.

Brugerflade

Der skal opstilles en brugerflade i form af et vindue, som ikke er en konsol denne har følgende specifikationer. Der skal opsættes et gitter således brugeren kan indsætte noder i form af lyskryds, bindepunkter fra vej til vej, vigepligt, huse og destinationer. På denne måde binder disse noder sig til en af kanterne i gitter systemet. Samtidig bliver det nemt at implementere A* algoritmen, og vejnettet bliver opstilt på en systamatisk måde. Brugeren får også et større overblik over vejnetværket, da brugeren skal kunne se hele gitteret oppefra.

Vejene skal implementeres således at de binder sig til noderne, så A* algoritmen kan beregne en vej igennem vejnetværket. Derudover giver det brugeren muligheden at tilkoble veje til lyskryds og vigepligte. Der skal også implementeres eksklusive veje, dette er en central del af simuleringen, da disse veje skal fungere således brugeren kan måle på hvordan trafikken ændre sig, hvis man tilføjer en eksklusiv vej. Udover det fungere de på samme måde som almindelige veje. De eksklusive veje er valgt at have med, så brugeren kan ændre på trafikken og se hvordan netværket udspiller sig, hvis man ændre på de allerede eksisterene veje.

Programmet skal også implementere lyskryds og vigepliger, fordi dette er en central del af et vejnetværk i den virkelige verden. Hvis ikke disse bliver implementeret så bliver programmet ikke nær så realistisk, som den virkelige verden. Der skal også indsættes, huse, destinationer og parkeringspladser i form af noder, husene indsættes så diverse køretøjer har et startpunkt og slutpunkt. Destinationerne indsættes så køretøjerne har individuelle destinationer, dvs køretøjet skal starte fra huset, og køre ud til en destination, senere på dagen skal køretøjet køre tilbage til huset. Disse destinationer skal også have en parkeringsplads, i form af en node. Disse parkeringspladser vil være et slutpunkt for rejsen fra køretøjets hus, hvor køretøjet søger efter den parkeringsplads der er nærmest destinationen. Dette er valgt, da billister i den realistiske verden, kan have destinationer uden parkeringspladser, som beskrevet i afsnittet med Altrans. Der skal også tilføjes fodgængerfelt, således køretøjet bremser, hvis en fodgænger vil forbi en vej. På denne måde skaber programmet et realistisk perspektiv i form af simulering. Samtidig skaber det også menneskelig adfærd i trafikken.

Simulering

Programmet skal indeholde en sammenligning af to simuleringer, dette vil foregå ved at brugeren kan opstille særlige veje, der kun bliver medtaget i en af simulationerene, og brugeren vil derefter kunne sammenligne outputtet. På denne måde kan brugeren sammenligne to simulationer og vurdere hvilken simulering som er den mest effektive, og derefter kan det udføres til den virkelige ver-

den. Der skal implementeres fodgængere, således det påvirker trafikken i form af fodgængerfelter. Programmet skal indeholde forskellige typer af køretøjer som, biler, busser, lastbilver osv. Dette er valgt, da programmet bliver mere realitisk af at indeholde forskellige typer, da køretøjerne accelerere og decelerere anderledes. Dette vil netop påvirke trafikken, samtidig køre nogle køretøjer langsommere og andre hurtigere. Dette skal opsættes således bruger selv definere et køretøj i programmet. Diverse køretøjer skal beregne den hurtigste vej til deres destination, da billister i dag foretrækker den hurtigste vej. Programmet skal implementere acceleration og deceleration på køretøjerne, dette er en central del af en simuleringen, da netop acceleration og deceleration kan skabe trafikpropper. Derudover er der også valgt, at implementere dette, da VisSims acceleration og deceleration ikke var præcise se afsnit om Teknologianalyse.

Program opbygning

Følgende afsnit beskriver hvordan opbygning af programmet forventets at blive, ud fra de forrige afsnit hvor de hovedsaglige komponenter er blevet beskrevet.

Programmet forventets at blive opdelt i 6 hovedområder som kan ses på figur 9.1.

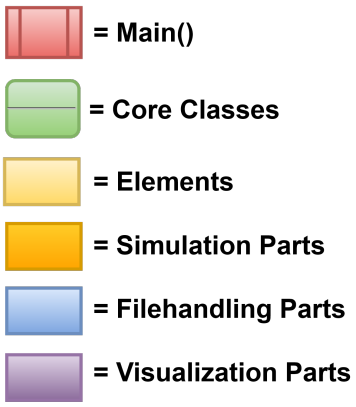


Figure 9.1: Opdeling

Main() vil være områder som fremstiller de hovedsaglige objekter til sammenkoblig af programmet.

Core Classes er de grundlæggende klasser som forbinder de mindre klasser med hiandnen. Disse klasser vil derfor stå få komunikationen mellem programmet.

Elements vil være de komponeter som bruges til simulation, samt visualsering, det kunne f.eks. være veje, parkeringspladser osv. Det er altså elementer som bruges til både at opretholde retningslinjerne for simuleringen, men også visualiseringen af simulationen.

Simulation Parts er de dele af programet som kun bruges til simuleringde-

len.

Filehandling Parts er de dele af programmet som kun bruges til håntering af filer, som f.eks projekt og simulations filer. Disse dele bruges altså til at sende generalle data fra forskelige dele, som så bruges af resten af programet.

Visualization Parts er de dele af programet som kun bruges til visualisere programmet og dens simulationer.

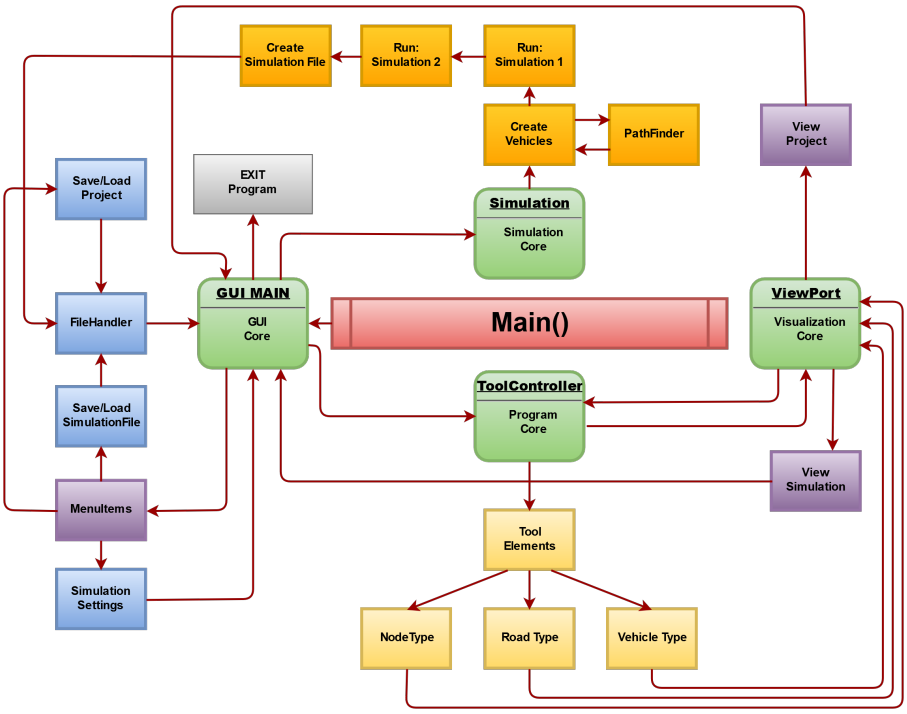


Figure 9.2: Program opbygning

På figur 9.2 ses hvordan vi forventer de 6 beskrevet dele arbejder sammen, for at opfylde vores krav til programet.

Programmets flow
Main() vil skulle kalde GUI MAIN når programmet states, da det er brugerens interface kerne, og vil derfor stå for alt input fra brugeren. GUI MAIN forbinder derfor også de resterende "core classes" da simulations delen ikke vil kunne køres uden input fra brugeren, da den f.eks skal kende til "Simulation Settings" som kommer fra Filehandler, som er forbundet med GUI MAIN.

Det samme gælder for ToolController som ikke vil kunne vælge/fremstille elementer uden brugeren fortæller hvilke elementer det er, og samme princip ses ved ViewPort, da den skal have input fra ToolController omkring hvilke

elementer som skal vise, samt ved indlæsning af filer, vil ViewPort skulle have data fra FIlehander, som igen skal køres igennem GUI MAIN.

Herved opnår vi en god nedkapslig af programmet, da der kun vil være 4 kerne klasser som har hvert deres formål, som beskrives nedenfor.

GUI MAIN skal som sagt initialisere de andre "core classes" ved opstart af programmet, derudover skal GUI MAIN håntere input fra brugeren. GUI MAIN står derfor for "MenuItems" som f. eks. save, load, settings osv. som vil være menupunkter for brugeren. Desuden vil GUIMAIN modtage inputs som valg af tilføje en vej(element). Dette input sendes efterfølgende til ToolControlleren.

Ud fra hvilke input ToolControlleren får fra GUI MAIN skal det tilsvarende element vælges f.eks en vej. Hvorefter ToolControlleren udfører handlingen til den valgte element i dette tilfælde tilføje en vej til projektet. Derefter sendes outputet fra ToolController til ViewPort. Da vejen kun er tilføjet i programmets data, men ikke visualiseret.

ViewPorten visualisere derfor elementet, altså i dette tilfælde at tegne vejen i projektet. Ved indlæsning af et projekt vil samme fremgangsmåde forkomme, GUIMAIN sender dataerne fra projekt filen, hvorefter ToolControlleren fremstiller alle elemnter, og ViewPorten vil til sidst visualisere alle elementer i projektet,

Alle dele af programmet vil i sidste fører tilbage til GUIMAIN som et loop, hvor brugeren så igen kan vælge sin handling, dette kunne f.eks. være at lave simulationen. Simulationen virker ved at fremstille køretøjerne ud fra de indstillinger som brugeren har sat, hvor køretøjets rute bliver beregnet i fremstillingsprocessen. Derefter køres den første simulation, og dernæst den anden. Hvor simulations resultaterne gemmes i en fil, som så kan køres i programmet og vise simulationen.

Programmet vil skulle blive ved med at gøre i et loop indtil at brugern lukker programmet.

Succeskriterier

Den følgende liste beskriver hvilke kriterier programmet skal opfylde, før programmet kan være en løsning til problemformuleringen. Der er lagt fokus på simulering af køretøjer som biler og busser. Andre simulerings enheder som fodgængere, cyklister og toge er udeladt da de har en mindre påvirkning på trafikken på vejnettet. Derudover er importering af kort ikke taget med som et succeskriterie, da det ikke er nødvendigt for at kunne opstille et vejnet. De udeladte elementer bliver diskuteret senere i rapporten.

1. Brugeren skal være i stand til at opsætte et vejnet, der indeholder objekter som trafiklys, huse, parkeringspladser og destinationer.

2. Det skal være muligt for brugeren at kunne gemme deres arbejde, lukke programmet og fortsætte deres arbejde næste gang de åbner programmet.

3. Simuleringen skal kunne sammenligne trafikafviklingen på vejnettet med og uden de eksklusive veje. Køretøjerne i de to simuleringer skal være de samme, så resultat ikke er tilfældigt.

4. Køretøjerne skal kunne finde den hurtigste rute til den parkeringsplads der ligger nærmest destinationen. Denne udregning skal tage højde for hastighedsgrænserne på vejene.

5. Køretøjernes bevælgse skal gøres realistisk med hensyn til acceleration og deceleration.

6. Programmet skal forklare knapperne og funktionerne i programmet, så det er nemt for brugeren at benytte programmet uden en manual.

10 Implementation

I dette kapitel vil vi beskrive implementeringen af de forskellige dele af løsningsforslaget. Vi har udvalgt nogle forskellige, vigtige dele, af det udviklede program og forklare i dette afsnit hvilken begrundelse til de individuelle dele af programmet og hvilken del det bringer til programmets helhed. Det er ikke alle dele af programmet der er fremhævet i dette afsnit, den fulde kildekode er afleveret sammen med rapporten til eksermination og er ikke dokumenteret i samme større detalje som delene fremhævet i dette afsnit.

10.1 Klassediagrammer

For at få et overblik over program delene der skulle til for at kunne løse problemet, er der løbende blevet opstillet klassediagrammer. Klassediagrammerne for brugerfladen er ikke vist her, men kan findes i bilag A.1 og A.2.

I diagrammerne betyder skråtekst at klassen er abstrakt, plus er et offentligt medlem, og understregning betyder at medlemmet er statisk. Hver klasse har tre kasser, den første indeholder klassens navn, den anden kasse består af klassens fields, og den sidste indeholder klassens properties, metoder og events.

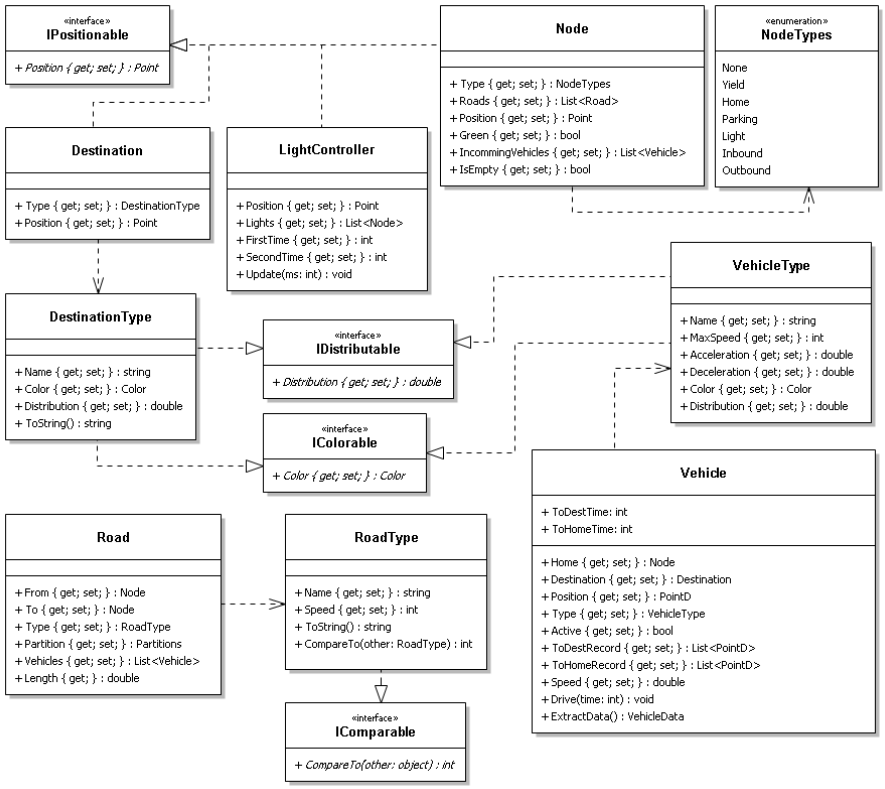


Figure 10.1: Elementer i vejnettet

Diagrammet der ses på figur 10.1 indeholder alle klasserne der er en del af vejnettet. `Node` klassen er de knuder vejne kan forbindes imellem. `Destination` og `LightControllerne` kan positioneres ligesom `Node` klassen, men de kan ikke blive forbundet til vejnettet. `Road` klassen beskriver vejene køretøjerne kan bevæge sig langs. `Vehicle` klassen beskriver et køretøj, og hvordan hastighed og bevægelsen skal foregå. `Destination`, `Road` og `Vehicle` klasserne har hver især en tilsvarende type-klasse, som brugeren kan lave nye instanser af og dermed bestemme elementernes egenskaber. Elementerne i dette diagram bliver uddybet på i afsnit 10.3, bortset fra `Vehicle` der bliver forklaret i afsnit 10.8.

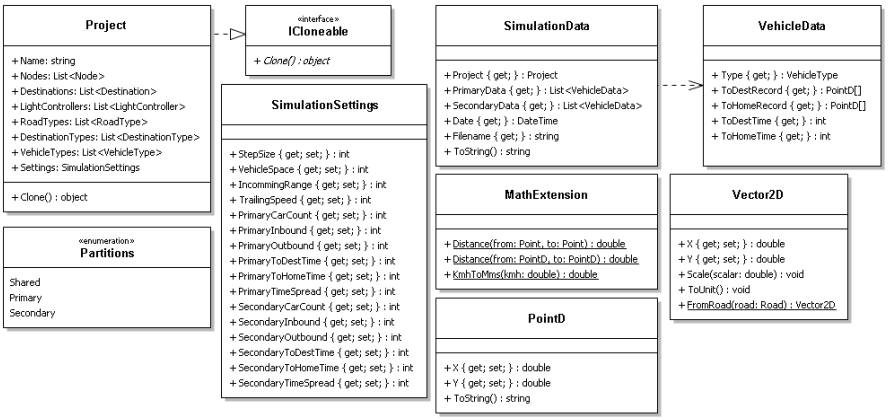


Figure 10.2: Diverse klasser

På figur 10.2 ses diagrammet for nogle forskellige klasser der ikke ligger under en bred kategori. `Project` klassen indeholder alt informationen om brugerens indstillinger og brugerens opbyggede vejnet. `SimulationData` indeholder en klon af projektet fra det tidspunkt det blev simuleret, og en optagelse af den positionelle data fra `Vehicle` instanserne der befærdede sig på vejnettet. `Partitions` er en enumerator der bliver brugt forskellige steder gennem programmet til at skelne mellem den primære og den sekundære simulering. `PointD` er en klasse der beskriver et punkt med doubles, for ikke at miste præcision ved at konvertere mellem floats og doubles. Den statiske klasse `MathExtension` indeholder nogle formler der ikke findes i standard biblioteket `Math`. `Vector2D` beskriver en vektor, og har nogle hjælpe metoder til at arbejde med vektorer. Disse klasser bliver uddybet i afsnit 10.4.

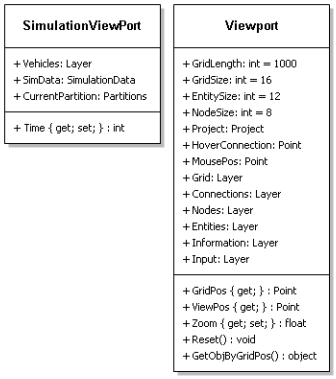


Figure 10.3: Viewport og SimulationViewport

De to klasser på figur 10.3 er brugerflade elementerne hvor brugeren kan se vejnettet. Den første klasse `Viewport`, er den der ses i programmets hoved brugerflade `GUIMain`, hvor der kan redigeres i vejnettet. Klassen `SimulationViewport` arver fra Viewport, og bruges til at vise hvordan køretøjerne bevæger sig. Disse

klasser forklares i afsnit 10.2.

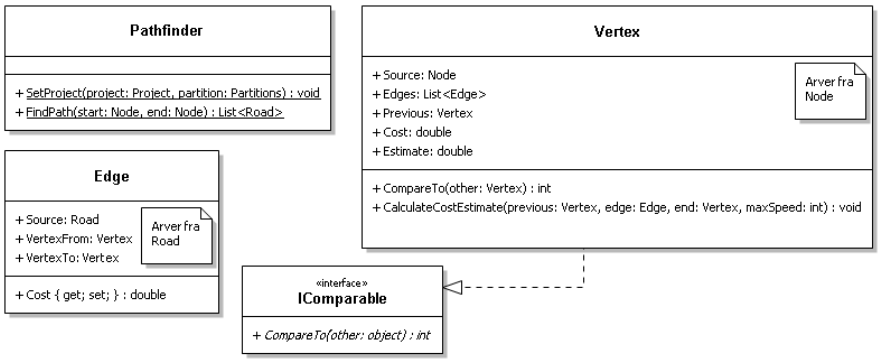


Figure 10.4: Pathfinder klassen

`Pathfinder` klassen, som vises sammen med `Vertex` og `Edge` klasserne på figur 10.4, bruges hver gang en `Vehicle` bliver konstrueret. Ved `Vehicle`'s konstruktion, findes den hurtigste vej til destinationen og ruten tilbage igen, som gemmes i selve `Vehicle` instansen. `Vertex` og `Edge` arver fra `Node` og `Road`, og indeholder yderligere informationer som `Pathfinder` bruger til at finde den optimale rute. `Pathfinder` beskrives i afsnit 10.6.

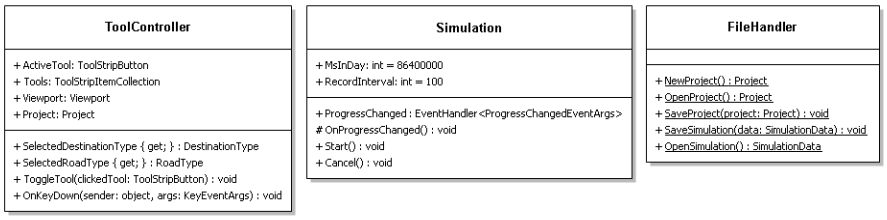


Figure 10.5: De funktionelle klasser

Diagrammet på figur 10.5 viser klasserne hvor en stor del af programmets logik bliver håndteret. `ToolControllerne` modtager input fra `GUIMain` når der bliver trykket på værktøjsknapperne, og modtager input fra Viewporten når der bliver trykket på gitteret, hvor den så derefter bestemmer hvad der skal ske baseret på det aktive værktøj. `Simulation` klassen håndterer selve simuleringerne af de primære og sekundære køretøjer, over en periode på 24 timer. `FileHandler` klassen er statisk og kan gemme og åbne `Project` klassen og `SimulationData` klassen. Disse klasser bliver forklaret i afsnit 10.7.

10.2 GUIMain

Dette afsnit omhandler flowet og indholdet af `GUIMain`, programmets start og hoved vindue.

På figur 10.6 ses `GUIMain`. GUIMain består af en menulinje(1), værktøjslinje(2) og en Viewport(3). Vinduet fungerer udelukkende ved brug af events.

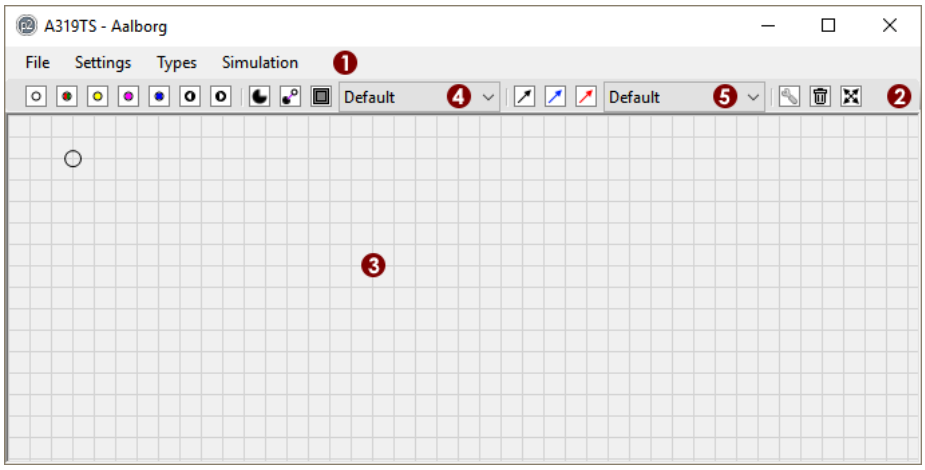


Figure 10.6: Programmets hoved vindue - GUIMain

Ved tryk på et menupunkt, vil der udløses en event, der åbner det tilsvarende vindue. Menupunkterne er delt op i 4 forskellige kategorier. `File` indeholder menupunkter der håndtrer fil åbning og gemning. `Settings` har indstillinger til det nuværende projekt, fordeling af destinationer og transportmiddelvalg, og sidst indstillinger for hvordan simuleringen skal udføres. `Types` har menupunkter til opsætning af forskellige destination, vej og køretøj typer. Sidst kan man igennem `Simulation` køre og vise simuleringer.

Værktøjslinjen har en række knapper der kan tjekkes. Når en knap bliver trykket på, bliver der sendt en event der kalder metoden `ToggleTool` på `ToolControlleren`. `ToggleTool` sørger for at der kun er en aktiv knap ad gangen. Derudover er der to lister på værktøjslinjen, i listen til venstre kan brugeren vælge den `DestinationType`(4) der bliver brugt, og i listen til højre kan brugeren vælge hvilken `RoadType`(5) der skal bruges.

`Viewporten` er gitteret hvor der er muligt at opstille et vejnet. `Viewporten` abonnerer på `Move` og `Click` begivenhederne. Hver gang brugeren flytter musen, vil `Viewporten` finde ud af hvor musen er, og tegne en cirkel, så brugeren kan være sikker på, hvilken gitter position der vil blive tilføjet til på forhånd. Ved tilfældet at brugeren trykker på `Viewporten`, tjekker `ToolControlleren` efter hvilket værktøj der er aktivt, og kører metoden der er forbundet til værktøjet.

10.3 Elementerne i Vejnettet

Dette afsnit beskriver de forskellige elementer der eksistere i vejnettet. Disse elementer kan bla. være knuder, destinationer, veje og lyskryds. Der vil i afsnittet blive gennemgået hvordan kildekoden er sat op for de forskellige elementer.

Node

I vores program anvender vi et grid hvorpå brugeren indsætter knuder der udgør de forskellige vejnet der bliver oprettet. Disse knuder kan være forskellige typer se nedenstående liste. Knudetyperne er opsat på en enumerator, således de kan tilgås igennem en variabel se figur 10.7. Dette er gjort således at processen i at oprette vejnet er relativt simple. Et simpelt eksempel ville være at brugeren indsætter to knuder, den første knude hvor brugeren ønsker køretøjerne skal køre fra, og en knude med typen `Parking` tæt på køretøjernes `Destination`. Således kan et meget simpelt vejnet opstille. Men det er også muligt for brugeren at opstille meget mere komplekse vejnet med lyskryds.

```
1 public enum NodeTypes { None, Yield, Home, Parking, Light, ↩
      Inbound, Outbound }
```

Figure 10.7: LightController klassen

None er en knude, som kan tilkobles mellem vejene.

Parking er knuden hvor køretøjerne parkere ved deres destinationer.

Home er startpunkt og slutpunkt for bilen.

Light er et lyskryds, hvor køretøjet skal standse hvis knuden er rød, ellers skal den køre videre, hvis den er grøn.

Inbound er en knude, hvor køretøjerne kommer fra og ind til vejnetværket i programmet.

Outbound er en knude, hvor køretøjerne køre ud fra, når et køretøj forlader vejnetværket.

Destination

`Destination` klassen i vores program er et punkt hvorpå køretøjerne vil søge henimod. Dette er ikke det punkt hvor køretøjerne stopper, til dette formål anvendes der en `Node` som er angivet til at være til parkering i nærheden af en `Destination`. `Destination` klassen består af en instans af `DestionationType` klassen, dette er, ligesom med `Vehicle` og `Road` klasserne, en klasse der bruges til at brugerdefinere forskellige typer af destinations med forskellige parametre, disse parametre kan ses på figur 10.8. Den første er `name` på en `Destination`, denne anvendes således brugeren kan have overblik over de forskellige destinationer. Derudover bruges `Color` til at vise farven på de forskellige destinationer, dette gøres også for at have overblik over destiationerne. Samtidig anvendes variablen `Distribution` til at definerer hvor mange procent af køretøjerne der køre til de individueller distinations typer.

```
1  public string Name { get; set; }
2  public Color Color { get; set; }
3  public double Distribution { get; set; }
4
5  public DestinationType(string name, Color color)
6  {
7     Name = name;
8     Color = color;
9     Distribution = 0;
10 }
```

Figure 10.8: LightController klassen

LightController

`LightController` klassen er den del af programmet hvori brugeren kan indstille på deres trafiklys knudernes opførsel.

```
1  public void Update(int ms)
2  {
3    _counter += ms;
4    if (_counter > _current)
5    {
6      if (_current == FirstTime)
7        _current = SecondTime;
8      else
9        _current = FirstTime;
10      ToggleLights();
11     _counter = 0;
12   }
13 }
14 private void ToggleLights()
15 {
16   foreach (Node light in Lights)
17   light.Green = !light.Green;
18 }
```

Figure 10.9: LightController klassen

`LightController` klassen har til formål kontrollere et trafiklys over tid. Dette er valgt at gøre ved brug af metoden `Update()`, se figur 10.9. I metoden tæller `_counter` op til den aktive tid, som er current, altså den tæller op til hvor lang tid der har været siden sidste lys skifte. Den selektivekontrolstruktur bliver udført, hvis `_counter` er over den tid vi venter på nu, hvor `_current` enten er `FirstTime` eller `SecondTime`. Inde i kontrolstrukturen bliver `_current` skiftet mellem `FirstTime` eller `SecondTime` således at lyskrydset skifter mellem rød og grøn, som ikke er visuelt. Metoden `ToggleLights()` kigger så igennem de knudetyper som er sat til `Light` og skifter dem, således de skifter mellem rød og grøn. Der er valgt at skifte lyskrydsene tidsbaseret, fordi flere lyskryds i den virkelige verden også er sat op til tid.

Road

Road klassen indeholder de variabler der skal anvendes for at kunne beskrive en vej i programmet.

```
1 public string Name { get; set; }
2 public int Speed { get; set; }
3
4 public RoadType(string name, int speed)
5 {
6   Name = name;
7   Speed = speed;
8 }
```

Figure 10.10: RoadType klassen

Road klassen tager imod en instans af `RoadType` som brugeren selv har defineret igennem `GUIMenuTypesRoads` klassen. På denne måde har brugeren kontrol over hvilken type vej der tale om og hvordan vejen opfører sig i programmet, dette kunne eks. være en motorvej. Vejene er programmeret således at brugeren selv kan opstille forskellige kryds, rundkørsler eller andre avanceret afkørsels baner. `RoadType` er en seperat klasse se figur 10.10 denne klasse definere to variabler som er `Name` og `Speed`. Disse to variabler sætter navnet på en vej og fartgrænsen på vejen. Ved brug af denne klasse, kan bruger så igennem `GUIMenuTypesRoads` overføre det data der er indtastet, og derefter bliver det gemt i `RoadType` klassen. Det er valgt at sættes dette op på denne måde for at skabe fleksibilitet af vejene.

10.4 Diverse

Dette afsnit dækker over klasserne der ikke ligger under en bred kategori af klasser.

Project

`Project` klassens formål er at holde på informationen om elementerne i vejnettet. Derudover holder `Project` klassen også på brugerens indstillinger for simuleringerne der udføres. Disse indstillinger er gemt i en instans af klassen `SimulationSettings`. Når `Project` klassen instansieres, tilføjer constructoren en `Default` type til `RoadTypes`, `VehicleTypes` og `DestinationTypes`, hvilket er listerne der gemme på alle typerne af deres slags. `Default` værdierne kan ikke slettes, da det ikke vil give mening for simuleringen at veje, køretøj og destinationer eksisterede uden en type, og programmet ville blive termineret.

```
1  public object Clone()
2  {
3    MemoryStream memory = new MemoryStream();
4    BinaryFormatter formatter = new BinaryFormatter();
5    formatter.Serialize(memory, this);
6    memory.Position = 0;
7    return formatter.Deserialize(memory);
8  }
```

Figure 10.11: Project.cs

`Project` klassen implementerer interfacet `ICloneable`, der beder om en implementation af `Clone()`, der returnerer en kopi af en instans af et `Project`. Implementationen af `Clone()` funktionaliteten er vist på figur 10.11, hvor instansen af `Project` bliver seraliseret vha. `BinaryFormatter` klassen, og derefter deserliseret samt returneret. Grunden til `Clone()` funktionen er at lave en 'deep copy' af projektet, da Simulation klassen har brug for en identisk men adskildt kopi, så køretøjerne i `Primary` og `Secondary` simuleringerne ikke kan se og holde for hinanden. En `MemberwiseClone()` som alle objekter har adgang til, laver kun en 'shallow copy', hvilket vil sige at knuderne og vejene stadig ville være referencer til de samme som i det første `Project`.

SimulationSettings

```
1  // Defaults
2  public SimulationSettings()
3  {
4    StepSize = 100;
5    VehicleSpace = 2;
6    IncommingRange = 10;
7    PrimaryCarCount = 1000;
8    PrimaryInbound = 100;
9    PrimaryOutbound = 100;
10   PrimaryToDestTime = 28800000; // 08:00
11   PrimaryToHomeTime = 57600000; // 16:00
12   PrimaryTimeSpread = 14400000; // 4h
13   SecondaryCarCount = 1000;
14   SecondaryInbound = 100;
15   SecondaryOutbound = 100;
16   SecondaryToDestTime = 28800000; // 08:00
17   SecondaryToHomeTime = 57600000; // 16:00
18   SecondaryTimeSpread = 14400000; // 4h
19 }
```

Figure 10.12: SimulationSettings

`SimulationSettings`, som kan ses på figur 10.12, indeholder alle indstillingerne der bliver brugt i en simulering. Constructoren til klassen sætter variablerne til de definerede værdier. Værdierne kan justeres gennem brugerfladen `GUIMenuSettingsSimulation`. I brugerfladen kan værdierne tilgås gennem nogle `NumericUpDown` elementer, hvor minimum og maximum er sat for at kontrollerer hvordan simuleringen kan indstilles. På tabel 10.1 kan begrænsningerne ses. Yderligere er der reglen at `TimeSpread` må ikke gøre det muligt at køretøjer

kan starte deres rejser uden for tidsrummet mellem 0 og 86400000 millisekunder. Der er lavet en brugerflade, der kan tilgås ved at trykke på en `Info` knap, som beskriver hvad variablerne betyder, da navnene muligvis kunne blive misforstået.

Variable	Minimum	Maximum
StepSize	10	100
VehicleSpace	0	100
IncomingRange	0	100
TrailingSpeed	0	100
CarCount	0	10000
Inbound	0	100
Outbound	0	100
ToDestTime	0	86400000
ToHomeTime	0	86400000
TimeSpread	0	86400000

Table 10.1: Begrænsninger for simulering indstillingerne

Data

`SimulationData` er en klasse der indeholder dataen fra køretøjerne (`VehicleData`) fra en `Primary` og `Secondary` simulering, og en kopi af projektet simuleringen blev kørt på. Det er denne klasse der bliver gemt i en fil efter simuleringen, så brugeren kan åbne den senere gennem `SimulationView` brugerfladen. `VehicleData` indeholder arrays af punkterne der blev optaget gennem simuleringen, tiden på starten af rejsen mod og fra destinationen, samt køretøjets type. Ved begge klasser sættes dataen ind gennem constructoren, og det kan derefter ikke ændres igen, da egenskaberne for variablerne er med private settere. Data klasserne kan ses på figur 10.13 og 10.14.

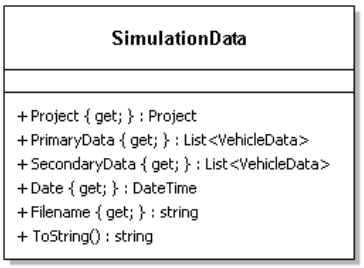


Figure 10.13: SimulationData klassen

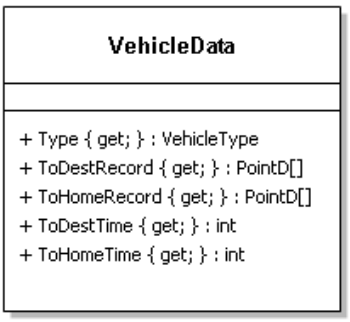


Figure 10.14: VehicleData klassen

MathExtension

`MathExtension` er en klasse der er lavet for at håndtere udregningerne fra `Point` til `Point`, fra `PointD` til `PointD` samt en kort kovertering af km/h til m/ms.

Der gøres bruge af klasser fra `Math` bibliotek. Måden vi udregner km/h til m/ms, er ved at tage km/h og dividere med 3600(60*60), se udregning 10.1. Vi benytter os af afstandsformlen i `MathExtension`, hvor vi beregner afstanden mellem to noder via deres koordinater 10.2. Afstandsformlen benyttes flere steder i programmet, men primært i `Pathfinder` og `Vehicle`, og konverteringen til meter på millisekund bliver kun bruger i `Vehicle.Drive()`.

$$km/h/3600 \tag{10.1}$$

$$\sqrt{(x2-x1)*(x2-x1)+(y2-y1)*(y2-y1)} \tag{10.2}$$

Vector2D

```
1  public void Scale(double scalar)
2  {
3    X *= scalar;
4    Y *= scalar;
5  }
6  public void ToUnit()
7  {
8    double magnitude = Length;
9    X /= magnitude;
10   Y /= magnitude;
11 }
12 public static Vector2D FromRoad(Road road)
13 {
14   Vector2D vector = new Vector2D();
15   vector.X = road.To.Position.X - road.From.Position.X;
16   vector.Y = road.To.Position.Y - road.From.Position.Y;
17   return vector;
18 }
```

Figure 10.15: Vector2D klasse

Formålet med `Vector2D` klassen er at gøre det nemt at flytte køretøjer, som bliver gjort i `TranslateVehicle()` metoden, der ligger i `Vehicle` klassen. `Vector2D` klassen har tre metoder, `Scale()`, `ToUnit()` og `FromRoad()`, som vises på figur 10.15. Den første metode `Scale()` ganger vektoren op med en scalar der bliver taget som input parameter. I `ToUnit()` bliver vektorens `X` og `Y` divideret med størrelsen af vektoren, således at vektorens størrelse bliver lig 1. `FromRoad()` metoden laver en `Vector2D` ud fra en `Road` instans koordinater.

10.5 Viewport

`Viewport` klassen har til formål at sætte rammerne for området hvori man kan tegne elementerne i vejnettet. `Viewport` er opsat til at fungere som en række af lag lagt oven på hinanden.

```
1 public Project Project;
2 public Point HoverConnection = new Point(-1, -1);
3 public Point MousePos = new Point(0, 0);
4 public Point GridPos { get { return GetGridPos(); } }
```

Figure 10.16

Klassen arver fra `Panel` klassen. Til dette formål er det nødvendigt for `Viewport` and være en del af et nyt projekt og derfor instansieres der en nyt `Project` og en række parametre set i figur 10.16. `Viewport` visualiserer alle enheder på panelet igennem de datasæt der eksisterer som en del af et `Project`.

`HoverConnection` visualiserer via en svævende streg, den forbindelse man prøver at lave mellem to objekter. `MousePos` indikerer det aktuelle punkt hvor musen befinder sig i gitter-systemet, selv ved en nedskalering vil den altid finde det samme koordinat. `GridPos` får data igennem en metode der indikerer alle mulige koordinater i gitteret. Dette er brugbart til at finde koordinaterne til objekterne der bliver tegnet i gitteret.

```
1  public object GetObjByGridPos()
2  {
3    Node node = Project.Nodes.Find(n =>
4                n.Position == GridPos);
5    if (node != null)
6        return node;
7    LightController controller = Project.LightControllers.Find(l =>
8                                 l.Position == GridPos);
9    if (controller != null)
10     return controller;
11   Destination dest = Project.Destinations.Find(d =>
12                      d.Position == GridPos);
13   if (dest != null)
14     return dest;
15   return null;
16 }
```

Figure 10.17

Som sagt kan `Viewport` indikere koordinaterne til indsatte objekter og dette gør den igennem metoden `GetObjByGridPos()` som set i figur 10.17. Metoden Tjekker for alle `Node`, `LightController` og `Destination` om deres position er lig `GridPos`. Hvis den har fundet en, så retunere den det fundne objekt. Grunden til dette er for at være i stand til at arrangere de håndterede objekter så deres koordinat kan identificeres.

Inde i `partial` klassen `ViewportSetup` bliver der angivet en række af lag (`Layer`) som set i figur 10.19. `Layer` er en klasse som arver fra `PictureBox` og har en constructor med `DoubleBuffered=true`. Grunden til at `PictureBox` ikke bare er benyttet alene er fordi måden disse `Layer` er anvendt i bl.a. `ToolController` klassen afhænger af at de kan genindlæses når et objekt, såsom en `Node`, skal skal ændres eller tegnes en ny. Dette gøres som vist i figur 10.18 fra `ToolController` klassen. Heri kan der ses at metoden til sidst genindlæser på `Nodes`. Hvis

`DoubleBuffered` sat til `false` ville programmet flimre visuelt, fordi objekter ikke bliver tegnet hurtigt nok til at virke omgående.

```
1  private void SetNodeType(NodeTypes type)
2  {
3      object obj = Viewport.GetObjByGridPos();
4      if (obj is Node)
5      {
6          if (type == NodeTypes.Light && ((Node)obj).Type == ↩
       NodeTypes.Light)
7              ((Node)obj).Green = !((Node)obj).Green;
8          else
9              ((Node)obj).Type = type;
10
11         Viewport.Nodes.Refresh();
12     }
13 }
```

Figure 10.18: Brug af Control.Refresh() i ToolController klassen

Disse lag indikerer i hvilken rækkefølge objekterne på panelet skal tegnes. Dette kan ses ved brugen af `Controls.Add` hvori der gradvist bliver lagt lag på lag. Hvis programmet ikke tegnede objekter i lag, så ville de alle skulle tegnes hver gang man udførte en enkelt opdatering, ligesom set i figur 10.18.

```
1  public Layer Grid = new Layer();
2  public Layer Connections = new Layer();
3  public Layer Nodes = new Layer();
4  public Layer Entities = new Layer();
5  public Layer Information = new Layer();
6  public Layer Input = new Layer();
7
8  Controls.Add(Grid);
9  Grid.Controls.Add(Connections);
10 Connections.Controls.Add(Nodes);
11 Nodes.Controls.Add(Entities);
12 Entities.Controls.Add(Information);
13 Information.Controls.Add(Input);
```

Figure 10.19: Lag for ViewportSetup, i deres kendetegnede rækkefølge

10.5.1 SimulationViewport

`SimulationViewport` har til formål at visualisere simuleringen, som `Simulation` klassen udfører. Klassen bruger den samme fremgangsmåde med lag som gennemgået tidligere.

SimulationViewport nedarver fra viewport. I 10.20 bliver der instansieret et nyt lag `Vehicles` og en variabel for `SimulationData` hvori indeholdes nødvendige datasæt for at udføre en simulation. `SimulationViewport` tager derudover `SimulationData` som en parameter til dens egen constructor som set i figur 10.21. I constructoren kan det ses at visualiseringen af simulationen benytter `Layer` til at fremstille `Vehicle` på `Viewport`. `Input` er et lag der kan

modtage nyt input og ved `Input.BringToFront()` angives der at der nu kan afventes nyt input.

```
1  private const int VehicleSize = 16;
2  public Layer Vehicles = new Layer();
3  public SimulationData SimData;
4  public Partitions CurrentPartition = Partitions.Primary;
```

Figure 10.20: Variabler til SimulationViewport

```
1  public SimulationViewport(SimulationData data) : base(data.Project)
2  {
3      SimData = data;
4
5      Grid.Paint -= DrawGrid;
6      Entities.Controls.Remove(Information);
7      Entities.Controls.Add(Vehicles);
8      Vehicles.Controls.Add(Input);
9      Input.BringToFront();
10 }
```

Figure 10.21: Constructoren til SimulationViewport

Til brug i selve simuleringen bliver der afsat en hel dag i millisekunder (`MsInDay` konstanten fra `Simulation` klassen). I `SimulationSettings` bliver der instansieret en variabel `Step` på 100 millisekunder der er afsat som standard, hvorfra denne også kan ændres i brugergrænsefladen. `Step` har til formål at opdatere den visuelle simulering som brugeren kan se, men inde i `Simulation` klassen er der en konstant ved navn `RecordInterval`. Denne variabel er hardcoded, kan ikke ændres af brugeren og indikerer hvornår simulering laver en optagelse af data.

Dette bliver der gået igennem i et **For-loop** som også findes i klassen `Simulation`. Simuleringen opdateres efter det antal som `Step` dikterer. `SimulationViewport` har så til formål at visualisere dette step.

```
1  private int _time = 0;
2  public int Time
3  {
4      get { return _time; }
5      set
6      {
7          if (value > MsInDay) _time = MsInDay;
8          else if (value < 0) _time = 0;
9          else _time = value;
10     }
11 }
```

Figure 10.22: Tids-afsætning for en simulering, MsInDay eller en hel dag.

10.6 Pathfinder

For at forklare implementationen af A*, tages der udgangspunkt i tre metoder som befinder sig i `Pathfinder` klassen, altså `FindPath`, `EstimateNeighbors` og `TracePath`. Grunden til at vi udvælger disse metoder er fordi de er vigtige i forhold til hvordan programmet skal finde frem til den hurtigste vej i programmet. Derudover vil der også kigges på en metode som ligger i `Vertex` klassen, der udregner kosten og estimeringen af den resterne kost.

```
1  public static List<Road> FindPath(Node start, Node end)
2  {
3    if (Vertices == null || start == null || end == null)
4      throw new ArgumentNullException();
5
6    InitLists();
7    SetStartEnd(start, end);
8    Start.Cost = 0;
9    Open.Add(Start);
10
11   Vertex current;
12   while (Open.Count > 0)
13   {
14     current = Open.Min();
15     if (current == End)
16     {
17       return TracePath();
18     }
19     else
20     {
21       MoveToClosed(current);
22       EstimateNeighbors(current);
23     }
24   }
25   throw new Exception("No path found");
26 }
```

Figure 10.23: FindPath metoden

Den første metode er `FindPath`, der ses på figur 10.23, hvor der startes med at overskrive listerne `Closed` og `Open` med tomme lister gennem metoden `InitLists`. `SetStartEnd` metoden som køres derefter, finder de `Vertex` der svarer til start og end noderne der bliver taget ind som parameter. I while løkken kigger den på `Open` listen og tjekker om der er nogle `Vertex` der ikke er evalueret endnu, hvis ikke kan ruten ikke findes og der kastes en exception. Inde i while loopet sættes det nuværende `Vertex` til at være den mindste på `Open` listen. Inde i `Vertex` klassen, er interfacet `IComparable` implementeret, således at `Min()` metoden returnerer den `Vertex` med den mindste `Estimate`. Hvis vi er ved enden returneres `Path` gennem metoden `TracePath`, ellers bliver vi ved med at vurdere naboerne til den nuværende `Vertex`, og sætter den nuværende over på `Closed` listen, så den ikke bliver vurderet igen senere.

```
1  private static void EstimateNeighbors(Vertex current)
2  {
3    foreach (Edge edge in current.Edges)
4    {
5      Vertex neighbor = edge.VertexTo;
6      if (!Open.Contains(neighbor) && !Closed.Contains(neighbor))
7      // Skip evaluated
8      {
9        Open.Add(neighbor);
10       double PossibleCost = current.Cost + edge.Cost;
11       if (neighbor.Cost > PossibleCost)
12       {
13         neighbor.Cost = PossibleCost;
14         neighbor.Previous = current;
15         double heuristic =
16           MathExtension.Distance(neighbor.Position, End.Position)
17           / MaxSpeed;
18         neighbor.Estimate = neighbor.Cost + heuristic;
19       }
20     }
21   }
22 }
```

Figure 10.24: EstimateNeighbors metoden

I `EstimateNeighbors` metoden, der kan ses på figur 10.24, kigger vi på naboerne, til den `Vertex` der bliver givet gennem parameteret. For hver nabo tjekkes der om naboen allerede er evalueret, hvis den er det bliver den sprunget over, ellers bliver den evalueret. Derefter tjekkes der om naboens mulige `Cost` fra den nuværende node, er bedre end den `Cost` der allerede er fundet. Hvis `PossibleCost` er bedre, bliver nabo knudens `Cost` og `Estimate` overskrevet, og `Previous` sættes til at være den nuværende `Node`, så ruten senere kan blive fundet igen.

```
1  private static List<Road> TracePath()
2  {
3    List<Road> roads = new List<Road>();
4    Vertex current = End;
5    while (current.Previous != null)
6    {
7      roads.Add(current.Previous.Edges.Find(edge =>
8                edge.VertexTo == current).Source);
9      current = current.Previous;
10   }
11   roads.Reverse();
12   return roads;
13 }
```

Figure 10.25: TracePath metoden

Sidst har vi på figur 10.25 metoden `TracePath`, der finder vejen tilbage, når algoritmen støder på slut punktet. Dette gøres ved at kigge på `Previous` referencen for den nuværende `Vertex`, og tilføje den vej der ligger mellem dem til en liste, ind til at `Previous` er lig `null`, hvilket vil sige at den er nået tilbage til start. Før ruten returneres bliver der kørt `Reverse()` på listen, så den står i

den rigtige rækkefølge.

10.7 Kerne Funktionalitet

For at håndtere alle dele af programmet blev der implementeret en et styringscenter, som heri findes som klassen `ToolController`. Formålet med denne fremgangsmåde skulle give bedre overblik over programmets mange metoder og give mulighed for let at håndtere fejl og ændringer i funktionaliteten. Klasserne i programmet er til dette formål håndteret objektorienteret og generelt nok til at fremgangsmåden kan fungere i praksis.

ToolController

`ToolController` klassen har til formål at forbinde de forskellige værktøjer så når brugeren f.eks. trykker på et af værktøjerne vil `ToolControlleren` kalde de tilsvarende metoder til værktøjet. Samt at der kun at være valgt et værktøj af gangen. `ToolController` er altså klassen som står for funktionerne som f.eks. `AddNode()`, `AddRoad()`, `AddLightController()` osv.

```
1  public ToolController(ToolStripItemCollection collection,
2                        Viewport viewport, Project project)
3  {
4     Tools = collection;
5     Viewport = viewport;
6     Viewport.Input.MouseClick += ViewportClick;
7     Project = project;
8  }
```

Figure 10.26: ToolController metoden

På figur 10.26 ses constructoren til `ToolController`, der bliver kaldt via GUIMain som sender alle værktøjerne, nuværende viewport samt projekt. Herved har `ToolController` alle elementerne til f.eks at tilføje en `Node`.

```
1  private void ViewportClick(object sender, MouseEventArgs args)
2  {
3    if (ActiveTool != null && args.Button == MouseButtons.Left)
4    {
5      switch (ActiveTool.Name)
6      {
7        ...
8        case "ToolAddNode": Add(typeof(Node)); break;
9        case "ToolLinkLight": LinkLight(); break;
10       case "ToolAddDestination": Add(typeof(Destination)); break;
11       case "ToolAddRoad": AddRoad(Partitions.Shared); break;
12       case "ToolPrimaryRoad": AddRoad(Partitions.Primary); break;
13       case "ToolEdit": Edit(); break;
14       ...
15     }
16   }
17 }
```

Figure 10.27: ViewportClick() metoden

Derudover bliver click eventen på `Input` laget af `Viewporten` i constructoren sat til at blive håndteret af metoden `ViewportClick()`. `ViewportClick()` der kan ses på figur 10.27 tjekker hvilket værktøj der er aktivt og kalder den tilsvarende metode.

```
1  public void ToggleTool(ToolStripButton clickedTool)
2  {
3    if (clickedTool.Checked)
4    {
5      clickedTool.Checked = false;
6      ActiveTool = null;
7    }
8    else
9    {
10     foreach (ToolStripButton tool in
11       Tools.OfType<ToolStripButton>())
12       tool.Checked = false;
13     clickedTool.Checked = true;
14     ActiveTool = clickedTool;
15   }
16   StopConnection();
17 }
```

Figure 10.28: ToggleTool metoden

For at sikre at der ikke er "valgt" flere værktøjer på samme tid, kaldes metoden på figur 10.28 `ToogleTool`, hvergang et værktøj bliver trykket på. Metoden fravælger alle `ToolStripButtons` i værktøjsListen, hvorefter det nuværende værktøj bliver sat til `true` (active). Derefter bliver det valgte værktøj sat over i `ActiveTool` variablen. Til sidst bliver `StopConnection()` kaldt, som er en metode til at nulstille værktøjets handling, så hvis man f.eks. har valgt `AddRoad` så vil `StopConnection()` sikre at det næste klik på gitteret vil tilføje vejens startpunkt og ikke slutpunkt.

Klassen indeholder som sagt alle værktøjerne og derfor indeholder klassen også en del metoder, derfor vil kun de mest væsentlige værktøjer blive beskrevet.

```
1  private void Add(Type type)
2  {
3    object obj = Viewport.GetObjByGridPos();
4    if (obj == null)
5    {
6      if (type == typeof(Node))
7      {
8        Project.Nodes.Add(new Node(Viewport.GridPos));
9        Viewport.Nodes.Refresh();
10     }
11     else if (type == typeof(Destination))
12     {
13       Project.Destinations.Add(new Destination(Viewport.GridPos,
14                                SelectedDestinationType));
15       Viewport.Entities.Refresh();
16     }
17     else if (type == typeof(LightController))
18     {
19       Project.LightControllers.Add(new
20         LightController(Viewport.GridPos));
21       Viewport.Entities.Refresh();
22     }
23   }
24   else if (obj is Node)
25   {
26     ((Node)obj).Type = NodeTypes.None;
27     Viewport.Nodes.Refresh();
28   }
29 }
```

Figure 10.29: Add metoden

`Add` metoden der ses på figur 10.29 benyttes til flere værktøjer som f.eks. at tilføje en `Node`, `LightController` eller `Destination`. Udfra typen som bliver sendt fra metodekaldet bestemmes hvilket objekt som skal tilføjes. Hvis den nuværende position i gitteret er en `Node` bliver `NodeTypen` sat til `None` og gitteret vil blive opdateret med `Refresh()`.

```
1  private void SetNodeType(NodeTypes type)
2  {
3    object obj = Viewport.GetObjByGridPos();
4    if (obj is Node)
5    {
6      if (type == NodeTypes.Light &&
7          ((Node)obj).Type == NodeTypes.Light)
8        ((Node)obj).Green = !((Node)obj).Green;
9      else
10       ((Node)obj).Type = type;
11     Viewport.Nodes.Refresh();
12   }
13 }
```

Figure 10.30: SetNodeType metoden

`SetNodeType()` som er vist på 10.30, benyttes til at give den enkelte `Node` en type som f.eks. `Light`, `Yield`, `Home`, `Parking` osv. Metoden modtager en `NodeType` som bliver bestemt fra `ViewportClick()`. Hvorefter den checker om

objektet på den nuværende position i gitteret er en `Node`. Hvis det er en `Node` vil `NodeTypen` blive sat til den modtaget type. Til sidst vil gitteret blive opdateret med `Refresh()`.

```
1  private void AddRoad(Partitions partition)
2  {
3   object obj = Viewport.GetObjByGridPos();
4   if (obj != null && obj is Node)
5   {
6      if (_firstNodeConnection)
7      {
8        _firstNode = (Node)obj;
9        _firstNodeConnection = false;
10       Viewport.HoverConnection = ((Node)obj).Position;
11     }
12     else
13     {
14       _firstNode.Roads.Add(new Road(_firstNode, (Node)obj,
15                                     SelectedRoadType, partition));
16       if (Control.ModifierKeys == Keys.Shift)
17       {
18         _firstNode = (Node)obj;
19         Viewport.HoverConnection = ((Node)obj).Position;
20       }
21       else
22       {
23         _firstNodeConnection = true;
24         Viewport.HoverConnection = new Point(-1, -1);
25       }
26       Viewport.Connections.Refresh();
27     }
28   }
29 }
```

Figure 10.31: AddRoad metoden

`AddRoad()` som kan ses på figur 10.31, bruges til at tilføje en vej mellem 2 noder, derfor checkes der først om object på den nuværrende positon i gitteret er en node. Hvis det er en node vil der blive checket om `_firstNodeConnection` er sket, altså om startpunktet til vejen er blevet valgt. Hvis `_firstNodeConnection` er `true`, betyder det at det ikke er sket, og noden på den nuværrendeposition i gitteret vil blive sat til `_firstNode`, og `_firstNodeConnection` vil blive `false`. Det betyder at næste gang brugeren trykker på en node i gitteret vil programmet vide at `_firstNode` er blevet sat, og derfor tilføjes der en vej mellem `_firstNode` og noden på den nuværrende position i gitteret.

Hvis brugern holder "Shift" nede imens, vil programmet sætte `_firstNode` til den nuværrende `Node` efter at der er blevet tilføjet en vej, da den `Node` vil være startpunktet for den næste vej. Det er en implementation som gør det nemmere og hurtigere for brugeren at tilføje veje.

FileHandler

`FileHandleren` er sat op så at man har mulighed for at lave et ny projekt, åbne og gemme projektet. Der er blevet dannet tre metoder som håndtere de tre

valg for brugeren, for at gøre det mest læsevenligt for dem der skal læse koden. `FileHandler` gør sig brug af `BinaryFormatter` for at gemme og åbne de forskellige objekter i binær form. Vi startede ud med at bruge `XMLSerializer`, da vi lavede `FileHandleren`. Vi stødte ind på nogle problemer da `XMLSerializer` skulle læse to objekter som har en reference til hinanden, og det skabte en circular reference som var årsagen til vores program crashede på daværende tidspunkt. Ved denne fejl skiftede vi til `BinaryFormatter`, da den er i stand til at håndtere en circular reference.

Metoden `NewProject` er meget simpel, den åbner et vindue med en `TextBox`, der beder om et navn til det nye projekt. Hvis et navn blev indtastet vil der så blive oprettet et nyt projekt med det navn, og det vil erstatte `CurrentProject` i `GUIMain`.

```
1  static public Project OpenProject()
2  {
3    FileStream file = null;
4    try
5    {
6      OpenFileDialog fileOpen = new OpenFileDialog();
7      fileOpen.Filter = "TSP Files|*.tsp";
8      if (fileOpen.ShowDialog() == DialogResult.OK)
9      {
10       BinaryFormatter formatter = new BinaryFormatter();
11       file = new FileStream(fileOpen.FileName, FileMode.Open);
12       return (Project)formatter.Deserialize(file);
13     }
14     return null;
15   }
16   catch (Exception e)
17   {
18     MessageBox.Show("Error: " + e.Message);
19     return null;
20   }
21   finally
22   {
23     if (file != null)
24     file.Close();
25   }
26 }
```

Figure 10.32: OpenProject metoden

Metoden `OpenProject`, der vises på figur 10.32, kan åbne et eksisterende projekt, når brugeren trykker på `Open` i `File` menuen. Denne metode benytter sig af `OpenFileDialog`, som ligger under `System.Windows.Forms`. Koden benytter sig af try-catch-finally, hvor den går ind i try fasen og filtrerer alle andre fil-typer væk som ikke er en TSP fil (traffic simulation project), hvis TSP filen er valgt så vil metoden deserialisere og åbne det gemte projekt op. Sidst vil finally lukke filen, så andre kan komme til.

I `FileHandler` klassen findes der også en `SaveProject()` metode som gemmer projektet, som brugeren har arbejdet på. `SaveProject()` metoden benytter den samme kode struktur som `OpenProject()` metoden, som ses på figur 10.32.

Metoderne minder meget om hinanden, hvor SaveProjekt gemmer istedet for at åbne et projekt. Til gemning og åbning af simuleringsdata er der metoderne `OpenSimulation()` og `SaveSimulation()`, der fungerer på samme måde som `OpenProject()` og `SaveProject()`.

Simulation

`Simulation` klassen indeholder funktionaliteten der bruges til at simulere. Klassen tager to `BackgroundWorker` instanser i brug til at køre simuleringerne, en til den primære simulering og en til den sekundære. `BackgroundWorker` er en klasse der kan køre en metode på en ny tråd, så programmet kan multitaske. Begge `Backgroundworker` instanser er sat til at køre `Simulate()`, og `SimulationCompleted()` når `Simulate()` er færdig.

```
1  public void Run()
2  {
3    // Find Inbound and Outbound Nodes
4
5    SetupVehicles();
6
7    Tuple<List<Vehicle>, Project, Partitions> primaryArguments;
8    primaryArguments = new Tuple<List<Vehicle>, Project, Partitions>
9      (_primaryVehicles, PrimaryProject, Partitions.Primary);
10   Tuple<List<Vehicle>, Project, Partitions> secondaryArguments;
11   secondaryArguments = new Tuple<List<Vehicle>,Project,Partitions>
12     (_secondaryVehicles, SecondaryProject, Partitions.Secondary);
13
14   PrimaryWorker.RunWorkerAsync(primaryArguments);
15   SecondaryWorker.RunWorkerAsync(secondaryArguments);
16 }
```

Figure 10.33: Run metoden

Den offentlige `Run()` metode, som set på figur 10.33, bruges til at starte simuleringen. Først bliver `SetupVehicles` kørt der fylder `_primaryVehicles` og `_secondaryVehicles` listerne med Vehicle instanser ud fra indstillingerne brugeren har sat. Derefter startes de to `BackgroundWorker`, hvor de får en liste af køretøjerne der skal simuleres, selve projektet køretøjerne befinder sig i, og en enumerator der viser hvilken simulering der køres. Udover `Run()`, er `Cancel()` også offentlig. `Cancel()` metoden kalder `CancelAsync()` på de to `BackgroundWorker` instanser.

```
for (int i = 0; i < MsInDay; i += Project.Settings.StepSize)
{
  if (worker.CancellationPending)
    break;
  if (i % onePercent == 0)
    worker.ReportProgress(i, i / onePercent + "% " + partition);
  foreach (LightController controller in project.LightControllers)
    controller.Update(project.Settings.StepSize);
  for (int j = 0; j < vehicleCount; j++)
    vehicles[j].Drive(i);
}
```

Figure 10.34: Forløkke i Simulate metoden

`Simulate()` metoden starter med at udpakke argumenterne, der blev sendt i en `Tuple`, hvorefter forløkken set på figur 10.34, bliver startet. For løkken er indkapslet i et try-catch statement, der stopper begge simuleringer i tilfælde af fejl, og rapporterer det til brugerfladen. Inde i forløkken på figur 10.34 tjekkes der først om brugeren har stoppet simuleringen, hvor forløkken så bliver brudt. Derefter tjekkes der om det er tid til at rapporterer fremdriften af loopet, hvilket gøres ved hvert lige procenttal. Brugerfladen `GUIMenuSimulationRun` der viser fremdriften, abonnerer på `ProgressChanged` begivenheden på begge `BackgroundWorker` instanser. Når `ReportProgress()` bliver brugt, sendes fremdriften samt en besked, der beskriver hvor langt simuleringen er nået, og om det er den primære eller sekundære simuleringen der er tale om. Efter dette bliver alle `LightController` instanser der findes i projektet opdateret, og sidst køres `Drive()` metoden på alle køretøjerne. `Drive()` metoden beskrives i afsnit 10.8.

```
private void SimulationCompleted(object sender,
  RunWorkerCompletedEventArgs args)
{
  if (PrimaryWorker.IsBusy || SecondaryWorker.IsBusy
      || args.Cancelled)
    return;
  else
  {
    List<VehicleData> primaryData = new List<VehicleData>();
    foreach (Vehicle vehicle in _primaryVehicles)
      primaryData.Add(vehicle.ExtractData());

    List<VehicleData> secondaryData = new List<VehicleData>();
      foreach (Vehicle vehicle in _secondaryVehicles)
      secondaryData.Add(vehicle.ExtractData());

    SimulationData data = new SimulationData(Project, primaryData,
                                             secondaryData);
    FileHandler.SaveSimulation(data);
    Filename = data.Filename;
    OnSimulationDone();
  }
}
```

Figure 10.35: SimulationCompleted metoden

`SimulationCompleted()`, som kan ses på figur 10.35, bliver kaldet en gang

for hver simuleringen. Simuleringen bliver derfor først gemt når begge `BackgroundWorker` instanser ikke er igang med noget, og derudover bliver der heller ikke gemt, hvis simuleringen er blevet afbrudt. Når begge simuleringer er færdige, bliver dataen lagt i en `SimulationData` instans, der gennem `FileHandler` bliver konverteret til binær, som senere kan åbnes og ses gennem `SimulationViewport` klassen. Sidst bliver `OnSimulationDone()` kaldt, der informerer brugerfladen om at simuleringerne er færdige.

10.8 Vehicle

`Vehicle` klassen representerer et køretøj i programmet, og indeholder den offentlige metode `Drive`, som `Simulation` klassen bruger til at køre alle bilerne.

```
1 public Vehicle(Project project, Node home, Destination dest,
2                VehicleType type, int toDestTime, int toHomeTime)
```

Figure 10.36: Vehicle Constructor parametre

Constructoren til `Vehicle` tager imod en række forskellige parametre som ses på figur 10.36, og udfører nogle opgaver, så køretøjet er klar til at køre. Som figur 10.36 viser, så tager klassen imod et `Project`, dette bliver ikke gemt i selve `Vehicle` klassen, men det bruges til at finde den nærmeste parkings plads, eller en tilfældig `Node` med typen `Outbound`. Efter at have fundet slut punktet, bliver `_toDestPath` og `_toHomePath` fundet gennem `Pathfinder` klassen. `toDestTime` er den tid køretøjet skal begynde at køre mod destinationen, og `toHomeTime` er den tid hvor den skal begynde at køre tilbage igen. Sidst sættes `bool` variablen `Active` til at være `false`.

```
1  public void Drive(int time)
2  {
3    if (!Active) CheckActive(time);
4    else
5    {
6      Speed = GetSpeed();
7      if (Speed != 0)
8        Move(MathExtension.KmhToMms(Speed) * _settings.StepSize);
9      if (time % Simulation.RecordInterval == 0
10         && !_toHomeStarted)
11       ToDestRecord.Add(new PointD(Position));
12     else if (time % Simulation.RecordInterval == 0
13              && _toHomeStarted)
14       ToHomeRecord.Add(new PointD(Position));
15   }
16 }
```

Figure 10.37: Drive metoden

Som afsnit 10.7 beskriver, bliver `Drive` kaldt for hver `Vehicle` instans, hver gang simulationen tager et step. I `Drive` metoden som vist på figur 10.37, bliver der først tjekket om køretøjet er aktivt, hvis ikke kaldes `CheckActive`, der ser om tiden er højere end det tidspunkt hvor køretøjet skal begynde at køre, og hvis

det er sandt vil køretøjet så aktiveres. I tilfældet at køretøjet allerede er aktivt, findes hastigheden bilen skal køre gennem metoden `GetSpeed`, og derefter hvis hastigheden ikke er nul, vil køretøjet flytte sig en afstand baseret på køretøjets nuværende hastighed. Efter køretøjet har bevæget sig, vil positionen blive gemt i enten `ToDestRecord` eller `ToHomeRecord`, alt efter hvilken rute der bliver kørt på tidspunktet. Positionen bliver kun gemt hver gang tiden kan gå lige op med konstaten `Simulation.RecordInterval`, det vil sige at positionen kun bliver gemt 10 gange i sekundet, selvom step størrelsen godt kunne være mindre. Grunden til at der ikke bare bliver gemt hver gang biler har flyttet sig, er at simulationen vil kræve alt for meget hukommelse.

```
1  private double GetSpeed()
2  {
3    int incomingVehiclesCount = _currentRoad.To
4                               .IncomingVehicles.Count;
5    Vehicle vehicleInfront = VehicleInfront(_settings.VehicleSpace);
6    if (CurrentNode != null
7        && CurrentNode.Type == NodeTypes.Light
8        && !CurrentNode.Green)
9      return 0;
10   else if (CurrentNode != null
11            && CurrentNode.Type == NodeTypes.Yield
12            && incomingVehiclesCount > 0)
13     return 0;
14   else if (vehicleInfront != null)
15   {
16     if (Type.MaxSpeed > vehicleInfront.Speed)
17       return vehicleInfront.Speed *
18              (_settings.TrailingSpeed / 100);
19     else return Type.MaxSpeed;
20   }
21   else
22   {
23     if (Type.MaxSpeed > _currentRoad.Type.Speed)
24       return _currentRoad.Type.Speed;
25     else return Type.MaxSpeed;
26   }
27 }
```

Figure 10.38: GetSpeed metoden

På figur 10.38 vises metoden `GetSpeed`, som var den første der blev kaldet i `Drive` metoden. Først tjekkes der på linje 6, om køretøjet er på en `Node`, (`CurrentNode` er null hvis køretøjet er på en vej mellem to noder), og hvis den `Node` så har typen `Light`, og at `bool` variablen `Green` er false, svarer det til at køretøjet er ved et rødt lys, og hastigheden bliver dermed returneret som et 0. Derefter tjekkes der på linje 10, om `Noden` har typen `Yield`. Hvis det er sandt, og at der er indkommende køretøjer som ikke er dette køretøj, så vil hastigheden også sættes til 0. Hvis ikke de to første er sande, så tjekkes der efter om der er et køretøj foran dette køretøj indenfor rækkeviden `VehicleSpace`, som brugeren kan indstille i settings. Hvis der findes et køretøj, og køretøjet kører langsomere end det nuværende køretøj, bliver hastigheden sat lig med køretøjet gange procentsatsen `TrailingSpeed`. Hvis køretøjet foran kører hurtigere, sættes hastigheden bare til køretøjets maksimale. Sidst har vi tilfældet

hvor der ikke er noget der blokerer køretøjet, hvor hastigheden vil blive sat lig hastighedsgrænsen på vejen, medmindre køretøjets egen max hastighed er lavere hvor hastigheden bare vil sættes til det maksimale.

```
1 private void Move(double distanceToMove)
2 {
3   CurrentNode = null;
4   TranslateVehicle(distanceToMove);
5   ControlOverreach();
6   ShowAsIncoming();
7 }
```

Figure 10.39: Move metoden

Figur 10.39 viser `Move` metoden. Som navnet indikerer, bruges metoden til at flytte køretøjet. Først bliver `CurrentNode` sat til null, da Move metoden aldrig bliver kaldet med en distance på 0, og vil dermed altid flytte sig fra den nuværende `Node`.

Derefter kaldes `TranslateVehicle`, hvilket er en metode der flytter køretøjet i retningen af vejen, uden at overveje om den har kørt for langt. Måden metoden gør dette på er illustreret på figur 10.40.

1. Et køretøj på en vej
2. Vejen laves om til en vektor
3. Vej-Vektoren laves til en unit vektor
4. Vektoren skaleres med afstanden der skal køres
5. Vektorens X og Y koordinater bliver adderet til positionen på køretøjet
6. Køretøjet har flyttet sig

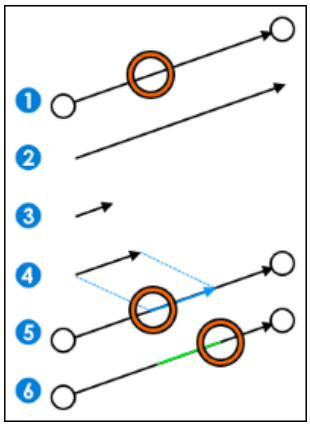


Figure 10.40: TranslateVehicle

```
1  private void ControlOverreach()
2  {
3    double currentRoadStartDistance = MathExtension
4      .Distance(Position, new PointD(_currentRoad.From.Position));
5    if (currentRoadStartDistance > _currentRoad.Length)
6    {
7      if (_currentPathIndex + 1 == _currentPath.Count)
8        Deactivate();
9      else if (_currentRoad.To.Type == NodeTypes.Light
10               || _currentRoad.To.Type == NodeTypes.Yield)
11       GoToNextRoad();
12     else
13     {
14       double remainingDistanceToMove = currentRoadStartDistance
15                                        - _currentRoad.Length;
16       GoToNextRoad();
17       Move(remainingDistanceToMove);
18     }
19   }
20 }
```

Figure 10.41: ControlOverreach metoden

Efter `TranslateVehicle` har flyttet køretøjet, bruges `ControlOverreach` til at tjekke om der er blevet kørt for langt, altså udover enden af vejen. Det findes ved at udregne distancen til knuden der ligger i starten af den nuværende vej `(CurrentRoad)`, og hvis den er længere end vejen så må køretøjet have kørt for langt. Når der er kørt for langt, tjekkes der først om køretøjet er ved slutningen af ruten, hvor den så vil deaktiverer, ellers hvis slutknuden på den nuværende vej er af typen `Light` eller `Yield`, standser køretøjet bare på punktet, hvor der så i næste step vil skulle tjekkes om den må køre videre. Sidst hvis køretøjet gerne må fortsætte, findes den afstand der er blevet kørt for langt med, køretøjet sættes ind på den næste `Road` på `CurrentPath`, og `Move` bliver kaldet igen.

Som det sidste der sker i `Move` metoden, har vi metoden `ShowAsIncomming`, der sætter køretøjet på en liste af indkommende biler på knuderne foran, hvis de er indenfor en afstand brugeren kan sætte i indstillingerne. Dette bruges når andre køretøj skal vurdere om de må køre ved en `Node` med typen `Yield` (vigeplight).

10.9 Unit Test Implementation

I dette program er testing implementeret i form af **Unit Tests** hvori koden er blevet implementeret og testing udført bagefter. Testingen er fokuseret på metoder der blev vurderet og prioriteret til at være nogle af de mest relevante at udføre testing på. Vurderingen har taget udgangspunkt i udregningsmetodik og koordinathåndtering da disse har den mest direkte indflydelse på programmets processer. Derudover er testmetoderne hovedsagligt udarbejdet efter den forståede problematik der kunne opstå ved den implementerede kode.

Testmetoderne og kildekoden er separeret i hver deres `namespace`, henholdsvist **A319TS** (hvori kildekoden findes) og **A319TS.Tests**. Begge disse

tilhørerer den samme `solution` også ved navn **A319TS**. For at **A319TS.Tests** `namespace` kan benytte metoder fra **A319TS** er der lavet en reference inde i `namespace` via **Visual Studios** indbyggede referencesystem.

Testklasserne er navngjort efter klassenavnet der bliver testet på efterfulgt af *Tests*, et eksempel på dette kan ses på figur 10.42. Testmetoderne er navngivet efter dette format: **MetodeNavn__TestForventetOpførsel**. Der findes bedre måder at navngive testmetoderne såsom det mere udbredte format: **MetodeNavn__StadieUnderTesting__**

```
1 [TestClass]
2 public class MathExtensionTests
```

Figure 10.42: Eksempel på Testklasse, MathExtensionTests

Derudover er formatet af testmetoderne kodet efter AAA(Arrange, Act, Assert) mønsteret. Mønsteret går generelt set ud på at at først opstille sit test miljø **(Arrange)**, dernæst den aktuelle kode der udfører testen **(Act)** og til sidst tjekket efter hvorledes testen afgjorte om den originalse metode fungerede efter forventede opførsel **(Assert)**. Et eksempel på dette kan ses på figur 10.43, hvori metoden `CompareTo` fra `RoadType` klassen bliver testet efter hvorledes metoden returnerer det rigtige resultat med forskellige `Speed` værdier.

```
1  [TestMethod]
2  public void ↩
       CompareTo_TestOfInputAndResultWhenParsedWithTwoDifferentSpeedValues()
3  {
4      // ARRANGE
5      var rt = new RoadType("Motorvej", 100);
6
7      var rt1 = new RoadType("MiscVej", 60);
8
9      // ACT
10     var result = rt1.CompareTo(rt1);
11     var result2 = rt.CompareTo(rt1);
12
13     // ASSERT
14     Assert.AreEqual(0, result);
15     Assert.AreEqual(1, result2);
16 }
```

Figure 10.43: Testmetoden CompareTo_TestOfInputAndResultWhenParsedWithTwoDifferentSpeedValues

Som gennemgået i afsnit 8.4 kan det være en fordel af arbejde så tidligt som muligt med testfasen for at undgå fremtidig programfejl. Dog som sagt blev dette ikke udført i dette projekt, hvilket ledte til at tidsforbruget på at kode de forskellige testmetoder var højere end hvad de ellers kunne have været, havde de været implementeret tidligere i programfasen.

I figur 10.44 kan der ses et eksempel på en implementeret testmetode der aftester metoden `Distance` fra klassen `MathExtension`. Metoden har det primære formål at udregne distancen mellem to koordinatsæt og dette bliver testet med en fejlmargen på *0.00001*, da udregningerne bearbejder en `double` og ender med

en del decimaler der ikke er afgørende for resultatets holdbarhed. Det kan ses i testmetoden at den er udviklet efter samme AAA format.

```
1  [TestMethod]
2  public void ↩
       Distance_TestToSeeIfCalulationIsCorrectWhenInputIsTwoRandomCoords()
3  {
4      Point pointA = new Point(6, 10);
5      Point pointB = new Point(2, 15);
6
7          double result = MathExtension.Distance(pointA, pointB);
8
9          Assert.AreEqual(6.40312, result, 0.00001);
10 }
```

Figure 10.44: Testmetoden Distance_TestToSeeIfCalculationIsCorrect

Implementationen af unit testing i dette projekt var med intentionen om at fuldføre en acceptabel testmetode til at afvikle løsningsmodellen til problemformulering.

10.10 Fejl og mangler i programmet

Ikke alle programmets kravspecifikationer er blevet implementeret med ønsket effekt. I et af kriterierne ønskes det, at simuleringen også kan foregå med cyklister og fodgængere. Dette er dog ikke tilfældet, programmet simulere kun med køretøjer, så derfor har vi valgt at tage dette med i videreudvikling. Programmet opfylder dog heller ikke ønsket om acceleration og deceleration dvs. programmet bliver mindre realistisk, da dette var en central del af kriterierne. Samtidig var dette også vigtigt fordi VisSims acceleration og deceleration udspillede sig lineær, meningen var at lave en acceleration og deceleration som udspillede sig eksponentielt, da dette var mere realistisk. I tabel 10.2 er de følgende fejl og mangler.

Funktioner	**Løsningsforslag**
Programmet implementere ikke Fodgængere	Skal implementeres på samme måde som køretøjerne, dog skal der lave en knude, der hedder fodgængerfelt.
Programmet implementerer ikke cyklister	Skal implementeres således der laves en separat cykelsti, som cyklerne kan cykle på. derudover skal vejene ændres, således cyklister og bliver kan være på en vej.
Programmet implementerer ikke acceleration	Step skal ændre sig på en bestemt måde, således at step forøges gradvist ud fra køretøjets acceleration. Derudover holde øje med om der findes et objekt foran den.

Programmet implementerer ikke deceleration	Køretøjerne skal kunne se yield fra en bestemt distance, så de kan nå at decellerere på samme måde som acceleration, bortset fra at de skal kunne bremse i stedet.
Programmets åbne og gemme funktionalitet bruger for meget tid	Programmets åbne og gemme funktionalitet skal forbedres ved brug af `Protobuf`.
TimeSpread, ToHomeTime, SetTime og ToDestinationtime skal ændres således brugeren ikke indtaster millisekunder	Det der står i tekstboksen skal ændres så der står klokkeslæt eller timer, derefter skal teksten konverteres til en integer og ganges op.
Programmets skal implementere menneskelig adfærd så bilerne ikke altid kører deres maksimale hastighed på vejene.	Lave en ny klasse `Driver`, hvor man kan indstille menneskelig adfærd.
Der skal implementeres UnitTest på alle klasser i kildekoden.	Implementer UnitTests.
Viewport går ud over kanterne i højre side og bunden.	Når programmet genererer gitteret, skal den tjekke programvinduets bredde og højde. Samt sørge for at Viewport positionen ikke kommer til at være større end hele størrelsen minus bredden og højden.
Samling og analyse af simuleringens data.	`SimulationData` skal gemme alt data som simuleringen har, ved at programmet skal kigge på de forskellige positioner. Hvis positionen er lig hinanden så står den stille, ellers skal den måle hastigheden.
Implementere en hjælpeside til information omkring programmets dele.	Lave en hjælpe knappe, som forklarer med tekst, hvordan programmet fungere.
I et lyskryds kan køretøjerne lave uvendinger.	Der skal laves en ny vej, som man kun må køre ind på fra bestemte knude.

Table 10.2: Fejl og mangler

Optimering af performance

Hastighed

Som sagt har programmet en svaghed når det kommer til performance. For altefter tickraten på simulationen, samt antal af biler osv vil brug af CPU og hukommelse øge markant. Det betyder at hastigheden på simulationen, og åbning af simulationensfilen kan tage fra få sekunder til flere timer alt efter de valgte simulationindstillinger.
Der blev lavet en performance profile som vidste at serialize og deserialize af vores simulations fil som tog det meste af simulation/view tiden. Desuden kan programmet ikke åbne filer, hvis de er over 1.2 GB.(**fordi VS ikke kan**

håndtere filer over 1.2GB)

Der blev derfor også kigget på andre metoder at serialize og deserialize simulation filerne istedet for at bruge XML. Der blev fundet frem til Protocol Buffer, og BinaryFormatter som iføgle flere tests siges at være hurtigere end XML [12][11][6]. Der blev derfor implementeret BinartFormatter, da det som XML er en del af NET Framework hvilket gjorde det en del nemmere at implementere, samt det understøttede klasser som Point og Color, hvliket Protocol Buffer ikke gjode som startard(Brug surrogate klasser til at konvetere typer som Point, Color til at være kompitabel til Protocol Buffer). Men selvom at BinaryFormatter er hurtigere end XML bliver det stadig overgået af Protocol Buffer som i nogen tests er 12x så hurtig iforhold til BinaryFormatter [6][1], og siden programmet stadig er langtsomt nå det kommer til serialize og deserialize kunne Protocol Buffer midske simulation/view tiden, og afhjælpe med programmet ikke terminere fordi filen enten er for stor, eller tager forlang tid at åbne.

Hukomelses Brug
Dog vil dette ikke afhjælpe på hukommelses forbruget, for at løse dette kunne vi optimere brugen af datatyper f.eks. bruge floats isetdet for doubles, hvliket vil halvere brugen af hukomelse hvergang vi beytter en double. Vi bruger f.eks double ved vores grid hvor vi bruger points med doubles værdier.

En anden måde at løse hukomelses problemet på, er ved brug af Memory-Mapped Files klassen, da den vil kunne splite filen op i en bestemt mægnde bytes både ved gemning/indlæsning. Herved vil programmet ikke gemme/indlæse hele simulations filen på hukomelsen på engang, men kun den mægnden bytes som Memory-Mapped streamen er sat til. Herved ungåes det at programmet alt efter størelsen på simulationen har et hukommelses forbrug på måske over 4000MB, hvliket i sidste ende vil terminere enten programmet eller computeren.

CPU forbrug
For at optimere CPU forbruget, vil en mulig løsnig være at få GPU'en til at tegne grafikken i programmet, istedet for CPU'en. Dette kan gøres ved brug af Windows Presentation Foundation (WPF). Fordelen ved dette er at den mægnde CPU som bliver sparet, vil kunne bruges på simulationen istedet, samt er GPU'en hurtigere til at tegne grafik end CPU'en. Derudover vil implemitering af flere "threads" kunne udnytte flere kerner i computeren, og herved øge simulationshastigheden, da vores program kun indeholder 2. Hvliket vil sige at f.eks. på en core i7 maskine med 8 kerner vil kun 25% kraft blive brugt.

11 Diskussion

Dette afsnit diskuterer hvorvidt den udviklede løsning, er en løsning på den opstillede problemformulering, samt afklarer om succeskriterierne i design afsnittet er opfyldt. Til sidst i afsnittet vil der diskuteres om hvorvidt vores program er realistisk.

Diskussion af problemformuleringen

Som det første der beskrives i problemformuleringen, har vi at nuværende simuleringsværktøjer enten ikke er brugervenlige, eller at de ikke er fleksible. Den udviklede løsning minder om værktøjet VisSim, der blev analyseret i afsnit 5, i det at programmet giver brugeren mulighed for at opstille et vilkårligt vejnetværk, og ændre på en række indstillinger for simuleringen. Det der gør gruppens værktøj anderledes fra VisSim er at det ikke er muligt at opstille andre simuleringer end trafik afviklings simuleringer. Denne afgrænsning betyder at VisSim er langt mere fleksibel, men det er også denne fleksibilitet der gør at VisSim ikke er brugervenlig. Udover afgrænsningen, er den udviklede løsning gjort brugervenlig ved hjælp af værktøjstips og etiketter, hvor brugeren kan læse om de forskellige funktioner af knapper og indstillinger. På grund af fokuset på trafik afvikling og informationerne der befinder sig i programmet, vurderer vi at gruppens løsning er mere brugervenlig end VisSim.

Problemformuleringen spørger dernæst hvordan et mesosimuleringsværktøj kan optimeres i forhold til vedligeholdse. I afsnit 4, der omhandlede ændringerne i konteksten, fandt vi at antallet af køretøjer var stigende, og at der skete ændringer i befolkningens valg af transportmidler. I gruppens program er det muligt at indstille antallet af køretøjer der skal simuleres, og programmet kan dermed tilpasses til stigningen af antallet af køretøjer. Det er også muligt for brugeren at opsætte forskellige køretøjs-typer, og indstille hvor mange af køretøjerne skal have de forskellige typer. Det er ikke muligt i programmet at simulere cyklister, hvilket betyder at i tilfælde hvor transportmiddelvalget for en del af befolkningen ændrer sig til cykler, vil man ikke kunne vise dette i programmet, udover at der vil være færre køretøjer på vejnettet.

Problemformuleringen beskriver også at der skal tages hensyn til billisternes adfærd, hvilket ikke bliver opfyldt af programmet. I programmet køre køretøjerne den maksimale hastighed, med mindre at køretøjer foran køre langsommere. I den virkelige værden svinger den ønskede hastighed fra billist til billist, og en implementation af adfærden ville dermed gøre simuleringen mere virkelighedsnær.

Samlet set er det gruppens mening at programmet er en løsning på problemformuleringen, i det at programmet blev videreudviklet og funktionalitet som cyklister og menneskelig adfærd blev implementeret.

Succeskriterier

Alle succeskriterierne er blevet løst udover kriterien om køretøjernes bevælgse skal gøres realistisk med hensyn til acceleration og deceleration. I gruppens program er acceleration og deceleration ikke blevet implementeret. Hvis et køretøj eksempelvis støder på et rødt lys, vil køretøjet stoppe øjeblikkeligt. Ligeledes når lyset skifter til grønt igen, vil køretøjet nå sin maksimale hastighed øjeblikketligt. Grunden til at acceleration og deceleration ikke er blevet implementeret, er at vigepligt og trafiklys blev prioriteret højere. Konsekvensen af denne mangel, er at køretøjerne vil bevæge sig gennem vejnettet hurtigere end det burde være muligt.

Realisme

For at programmet skulle kunne bruges til at lave beslutninger, kræves det af programmmet, at resultatet af simuleringen er realistisk. Udover problemerne beskrevet herunder, er der også manglen på acceleration, deceleration og menneskelig adfærd som blev beskrevet i afsnit 11 og 11.

- Køretøjerne i simuleringen kan lave uvendiger i alle lyskryds, dette er ikke realistisk, da det ikke er lovligt i alle lyskryds.
- Vejnetværket er opstillet såleves en vej er et spor for en bil, dette gør også programmet urealistisk, da en vej kan indholde flere spor.
- Køretøjerne kan ikke overhale andre køretøjer, hvilket også er betydelig del af programmet.
- I programmet kan brugeren opstille et tidsrum hvor køretøjerne kører fra deres hjem til destinationen, og et tidsrum hvor der bliver kørt hjem igen. Dette fungerer ved at brugeren sætter et bestemt tidspunkt, og derefter indstiller hvor meget køretøjerne må afvige fra dette tidspunkt. Denne metode er urealistisk, da brugeren ikke kan specificere at der skal være meget trafik ved 8 tiden, og sammentidigt have en lav mængde trafik om natten og midt på dagen. Det ville være en forbedring, hvis brugeren kunne indstille procentvis antallet af køretøjer for mindre tidsrum gennem en dag, eksempelvis fra klokken 8 til klokken 9.

Dog er programmet realistisk på synspunkter så som, vigepligt, lyskryds, køretøjerne ikke køre ind i hinanden, køretøjerner holder en afstand til de andre køretøjer og hastighedsgrænser.

12 Konklusion

Det daglige tidsspild på 25 tusinde timer på de danske veje, som koster den danske befolkning mere end 2500 millioner danske kroner om året, er et problem hvor vi, på baggrund af problemananlysen, kan konkluderer at en mulig løsning ligger i optimeringen af det danske vejnet. En optimering af vejnettet vil kræve en udbyggelse ved alle de knudepunkter, hvor trafikken ofte bliver hæmmet eller går i stå, for at mindske ruternes belastning. Derfor er det vigtigt at den tiltænkte udbyggelse der skal laves for at aflaste ruterne, faktisk også vil afhjælpe problemet før der investeres i at udvide vejnettet. Til dette formål anvendes der simulering for at undersøge hvorvidt disse udbyggelser faktisk også vil gøre en forskel og hvorvidt det kan betale sig. Men ud fra vores analyse af eksisterende løsninger kan vi konkluderer at disse løsninger ikke er fleksible nok til at kunne tilpasse sig ændringer i konteksten eller er ikke særlig brugervenlige. Derfor er der blevet udviklet et simuleringsprogram, hvor det skulle have været muligt at afgøre, hvorvidt en ny rute eller udvidelse af vejnettet vil være en investering som kan betale sig at implementere.

Dette simuleringsværktøj gør det muligt for dem som skal træffe disse beslutninger, at teste deres løsninger, før de bliver implementeret i virkeligheden og på en billig og realitisk måde. Trods simuleringsværktøjet ikke er færdigudviklet, har det nogle fordele som de nuværende og tilgængelige simuleringsværktøjer ikke har, såsom at det netop kan tilpasse sig ændringer i konteksten i form af justering i vejnet, mængden af biler og mere, samt at det har den brugervenlighed der gør det nemmere at bruge.

På grund af vores fokus på at kunne præsenterer en realistisk simulering af trafik, har vi dog ikke haft success med at udvikle de funktionaliteter som skulle anvendes til at sammenligne data fra to forskellige simuleringer og i det hele taget kunne retunerer data fra simulering som output. Der kan derfor konkluderes at løsningforslaget ikke lever fuldt op til problemformuleringen da det ikke løser hele problemet på grund af manglen på resultater, manglen på evnen til at justerer bilernes adfærd og på samme tid lever det ikke op til alle succeskriterier.

13

Perspektivering

Vi har igennem programmet benyttet os af Window Forms for at kunne fremstille et GUI, som brugeren kan benytte sig af. Problemet er bare at når programmet skal tegne simuleringen, og når glitteret skal tegnes, og dette bliver gjort hver gang noget rykker sig på glitteret. Dette bliver gjort på cpu'en, hvilket bruger meget kraft og ressourcer. Istedet kunne man taget Windows Presentation Foundation (WPF) i brug, da WPF benytter sig af GPU'en når der skal tegnes, og så skal processeren kun tænke over det input der kommer fra brugeren. Dette vil kræve de samme resourcer at lave en simulering, da denne kører på cpuen. det vil dog gøre at opdateringer af grafik bliver hurtigere og ikke påvirker responstiden fra input lige så meget.

Vores program ville være bedre hvis vi benyttede bedre multithreading, da vores program kører på to tråde ved simuleringsdelen. Vi bruger kun 50 procent af en CPU på 4 kerner og 25 procent på en med 8 kerner. Hvis vi havde håndterert at kunne benytte 100 procent af en cpu, ved f.eks. bedre multithreading såsom et variabelt antal tråde der var igang, så vi fuldt ud benyttede af de ressourcer CPU'en har.

13.1 Analysering af simulationsoutput

På nuværende tidspunkt er simuleringsprogrammet ikke i stand til at give et tekstbaseret output, hvor man kan se hvordan implantationen af en ny strækning vil optimere på trafikken, eller hvor det vil være mest optimalt at implementere en ny strækning, for at optimere trafikken mest muligt. Det næste skridt i udviklingen af programmet, vil være at implementere en beregningsalgoritme, som ikke blot kan beregne, men også analysere, hvor det er mest optimalt at placere en vej, for at effektivere trafikken. Outputtet skal selvfølgelig være tekstbaseret, eftersom det vil være besværligt, for ikke at sige umuligt, at beregne blot

13.2 Testing

Som gennemgået i tidligere afsnit vedrørende testing så implementerede vi kun den funktionelle testing metode Unit Testing. Programmet er ikke udviklet i et **TDD**(Test-driven development) format. Problematikken ved ikke at udføre denne udviklingsmetode giver mulig anledning til fejltagelser ved kodeimplementationen. **TDD** er en udviklingsmetode orienteret omkring udvikling gennem testing. Ideen ligger i at kode tests før selve kildekoden bliver implementeret. Fordelen ved dette er at man går ind i kode-delen med en præcis viden om hvad metoden skal gøre.[18] Der ville have været mulighed for mere gennemgående testingformater, havde programfasen været udviklet i dette format.

Efter nøjere overvejelser over andre eksisterende testing metoder såsom Usability Testing og System Testing er der vurderet, at i forhold til at assistere løsningsmodellen for problemformuleringen, kunne have været ideelt at udføre en Usability Test på programmet. Testen handler i kort forstand en ren brugerbaseret interaktionstest. Brugeren benytter programmet og deres respons bliver taget til behandling af udviklerne.[21] Da dette projekts program skulle være brugervenligt ville dette være en ideel testmetode at udføre.

Bibliography

[1] protobuf-net not faster than binary serialization? http://stackoverflow.com/questions/2966500/protobuf-net-not-faster-than-binary-serialization.

[2] Pshko Aziz. Aalborg kommune teknisk forvaltning, 2016.

[3] Christopher A. Chung. *Simulation Handbook - A Prictical Approch.* CRC Press LLC, 2004.

[4] Parallel and Sequential Data Structures and Algorithms. http://www.cs.cmu.edu/afs/cs/academic/class/15210-f12/www/lectures/lecture13.pdf, 2012. Accessed: 22-05-2016.

[5] Graphs. http://www.csse.monash.edu.au/~lloyd/tildeAlgDS/Graph/, 2016. Accessed: 22-05-2016.

[6] Yan Cui. Performance test - binaryformatter vs protobuf - net. http://theburningmonk.com/2011/08/performance-test-binaryformatter-vs-protobuf-net/, 2011.

[7] Altair Engineering. About visual solutions, incorporated. http://www.vissim.com/company.html, 2016.

[8] Carsten Broder Hansen. Dtu - forskning. http://www.transport.dtu.dk/Forskning, 2015.

[9] Danmarks Miljøundersøgelser. Altrans - adfærdsmodel for persontrafik, faglig rapport fra dmu nr. 348. http://www.dmu.dk/1_viden/2_Publikationer/3_fagrapporter/rapporter/fr348.pdf, 2001.

[10] Danmarks Miljøundersøgelser. Modelanalyser af mobilitet og miljø. slutrapport fra altrans og amor ii, faglig rapport fra dmu nr. 447. http://www.dmu.dk/1_viden/2_publikationer/3_fagrapporter/rapporter/FR447.pdf, 2003.

[11] Youssef Moussaoui. Comparing the performance of .net serializers. https://blogs.msdn.microsoft.com/youssefm/2009/07/10/comparing-the-performance-of-net-serializers/, 2009.

[12] MAXIM NOVAK. Serialization performance comparison. http://maxondev.com/serialization-performance-comparison-c-net-formats-frameworks-xmldatacontractserialize Ukendt.

[13] Dwight A Hennessy og David L Wiesenthal. Traffic congestion, driver stress, and driver aggression. https://www.researchgate.net/profile/Dwight_Hennessy/publication/229863510_Traffic_congestion_driver_stress_and_driver_aggression/links/0deec53274dd4c9e88000000.pdf, 1999.

[14] Matthew Barth og Kanok Boriboonsomsin. Traffic congestion and greenhouse gases. https://escholarship.org/uc/item/3vz7t3db, 2009.

[15] Michael Knørr Skov og Karsten Sten Pedersen. Trafikprop. flere veje vil skabe større vækst. http://www.cowi.dk/menu/tema/infrastruktur-2030/cowi-i-medierne/Documents/Veje%20skaber%20v%C3%A6kst_Politiken%20analyse%2024052014.pdf, 2014.

[16] Anders Pihlkjær. Trafiksimulering med vissim. Technical Report 6, Aalborg Universitet - Vej og Trafikteknik, Marts 2009.

[17] Kenneth H. Rosen. *Discrete Mathematics and its Applications*. McGraw Hill, 7. global edition, 20113.

[18] The Benefits of Unit Testing. http://www.seguetech.com/blog/2014/10/10/benefits-unit-testing, 2014. Accessed: 22-05-2016.

[19] Unit Test naming best practices. http://stackoverflow.com/questions/155436/unit-test-naming-best-practices, 2013. Accessed: 22-05-2016.

[20] Danmarks TransportForskning. Trafikmodeller arbejdsnotat til infrastrukturkommissionen notat 3. http://www.transport.dtu.dk/~/media/Institutter/Transport/forskning/publikationer/publikationer%20dtf/2007/arbejdsnotat_om_trafikmodeller_160507.ashx?la=da, 2007.

[21] Usability Testing. http://www.usability.gov/how-to-and-tools/methods/usability-testing.html, 2016. Accessed: 23-05-2016.

[22] Vejdirektoratet. Længden af offentlige veje. http://www.vejdirektoratet.dk/DA/viden_og_data/statistik/vejeneital/l%C3%A6ngdeoffentligeveje/Sider/default.aspx, 2016.

[23] Vissim.com. http://www.vissim.com/downloads/doc/VisSim_UGv80.pdf, 2015. Accessed: 17-03-2016.

A

Appendix

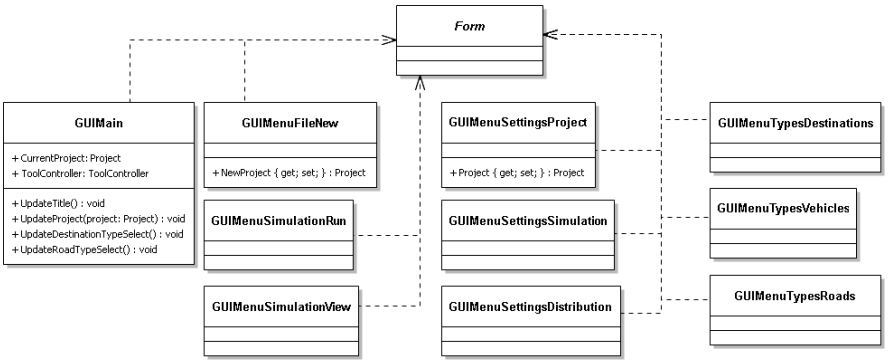


Figure A.1: Appendix A

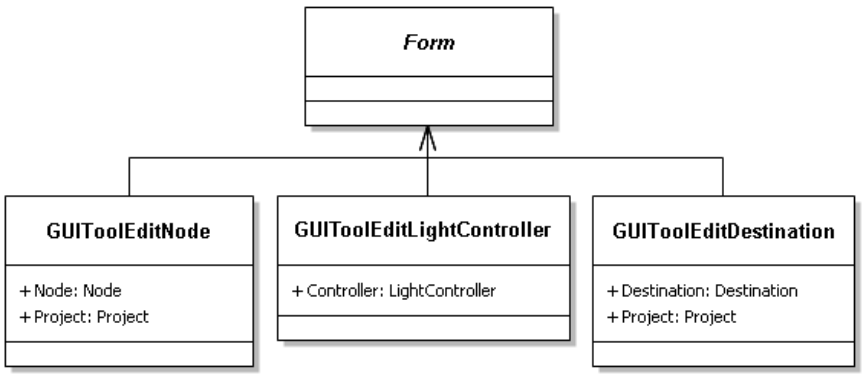


Figure A.2: Appendix B