
Simulering - Trafik - Temporary

Project Report
Group A319

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



AALBORG UNIVERSITY
STUDENT REPORT

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

Title:

Simulering - Trafik - Temporary

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: 68

Date of Completion:

May 14, 2016

Abstract:

Pasta ipsum dolor sit amet rotini
pasta al ceppo lagane spaghetti
penne lisce tagliatelle conchiglie.
Stringozzi ricciutelle capellini lasag-
nette pennoni lasagnette trenette
croxetti capelli d'angelo mafalde
farfalle ziti strozzapreti rotini. Tagli-
atelle sacchetti pasta al ceppo
spaghetti foglie d'ulivo capunti
tortiglioni vermicelloni fettuccine.
Penne zita gnocchi manicotti sac-
chetti fiorentine corzetti pasta al
ceppo stringozzi vermicelli fusilli
lanterne sacchetti fettucelle. Fiori
tuffoli fiori tuffoli capelli d'angelo
sagnarelli chifferi tuffoli ricciolini
cavatappi.

Benjamin Jhaf Madsen
<bjma15@student.aau.dk>

Kim Larsen
<klars15@student.aau.dk>

Jacob Sloth Thomsen
<jsth15@student.aau.dk>

Lasse Fisker Olesen
<lolese15@student.aau.dk>

Alexander Umnus
<aumnus14@student.aau.dk>

Niclas Struntze Bach
<nbach14@student.aau.dk>

Rasmus Thomsen
<rkth15@student.aau.dk>

Forord

Læsevejledning

Terminologi

Contents

Preface	iii
1 Indledning	1
1.1 Problemets Relevans	1
1.2 Initierende problemstilling	2
2 Simulering	3
3 Metode	6
3.1 Problemområde	6
3.2 Fremgangsmåde	7
3.2.1 Emperiske-induktive-metode	8
3.2.2 Andre metoder	8
3.2.3 Prøve-fejl-metoden	8
3.3 Kildekritik	8
I Problemanalyse	9
4 Ændring af Konteksten	10
4.1 Transportmiddelvalg	11
4.2 Eksisterende Modeller	11
5 Teknologianalyse	13
5.1 VisSim	13
5.1.1 Hvordan virker VisSim?	13
5.1.2 VisSim - Bilen	14
5.1.3 VisSim - Netværket	14
5.1.4 Analyse af acceleration og deceleration	15
5.1.5 Vedligeholdelse af VisSim	16
5.2 Altrans	16
5.2.1 Geografisk model	16

5.2.2	Adfærds model	17
5.2.3	Emissions model	18
5.3	Vurdering	19
6	Interessantanalyse	20
6.1	Transport- og Bygningsministeriets (TRM) - Vejdirektoratet . . .	20
6.2	Den Kommunale Sektor	21
6.2.1	Sim Firmaer	21
6.3	Uddannelsessektoren	21
6.4	Specificering af målgruppe	22
7	Problemformulering	23
7.1	Ny	23
II	Problemløsning	24
8	Teori	25
8.1	Tidskompleksitet	25
8.1.1	Tidskompleksitet af iterative kontrolstruktur	25
8.2	Grafteori	26
8.3	Dijkstras Algoritme	27
8.4	A* Algoritmen	29
9	Kravspecifikationer	33
9.0.1	Succeskriterier	34
9.1	Black Box Test - Unit Test	35
10	Implementation	37
10.1	Klassediagrammer	37
10.2	Hoved Brugerfladen	43
10.3	Elementerne i Vejnettet	44
10.3.1	Node	44
10.3.2	Destination	44
10.3.3	LightController	44
10.3.4	Road	45
10.3.5	Typer	45
10.4	Diverse	45
10.5	Project	45
10.6	SimulationSettings	46
10.6.1	Data	47
10.6.2	MathExtension	48
10.6.3	Vector2D	48
10.7	Viewport	49
10.8	Pathfinder	51
10.9	Funktionelle	53
10.9.1	ToolController	53
10.9.2	FileHandler	57
10.9.3	Simulation	58
10.10	Vehicle	58

11 Diskussion	63
12 Konklusion	64
13 Perspektivering	65
Bibliography	66
A Appendix	68

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 [15, 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.

Er man uheldig, kan man 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 [9]. Vejprojekterne inkluderer 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 af CO₂ forurening. Mængden af CO₂ der bliver udsluppet, afhænger af hastigheden billisterne kører. Ved en lav hastighed

kan CO₂ 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 brændstoffet [8, 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 aggressiv kørsel og i værste tilfælde ender det med en ulykke [7, s. 2-3].

For den almindelige bilist, så kan trafikpropper være irriterende at skulle igennem, da det er tidskrævende. Men når det kommer til ambulancernes udrykning, og det kan have fatale konsekvenser for nogle patienter. Falck har oplyst at det koster ambulancerne 1-2 minutter i udrykningstid, når der er trafikprop. Konsekvenserne kan variere alt efter hvor alvorligt syg patienten er, og i værste tilfælde så er konsekvensen menneskeliv [10].

1.2 Initierende problemstilling

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 variabler bruges til at simulere trafik?*
- *Hvem har gavn af disse trafikmodellerings simulatorer?*
- *Hvordan fungerer de forskellige eksisterende software, trafik modellerings simulatorer?*

Simuleringsmodellering og analyse er en del af processen, når man skal programmere et 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. Normalt vil ved hjælp af dataene til at kunne analysere nogle forskellige problemer i virkeligheden for at kunne foretage sig vigtige valg inden for drift- eller politiske ressource beslutninger. Man kan også benytte simuleringsmodel til at træne folk i at tage bedre beslutninger eller forbedre ens egen ydeevne inden for et område. [2, s. 16-20]

En anden form for computer simulation er computer simulator. Simulatorer er også baseret på eksisterende eller foreslået system, ligesom simulationsmodel. I modsætning til simulationsmodel, hvor man laver valgene på forhånd, så laver man valgene imens simulatoren er i gang. Meningen med simulatoren 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 systemet 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 i simulation fremfor i virkeligheden
- Det er blevet lettere at analyse et system
- Nemt at demonstrere hvordan modellen fungerer

Hurtigere resultater i simulation fremfor i virkeligheden Igennem simulering af et system kan man bestemme nogle variabler, det kan være sådan noget som 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 blevet lettere at analyse et system Før man kunne benytte computeren til at lave en simulation over et system, så krævede det mange ressourcer at kunne analyse et problem. Hvis man skulle have en meget kompleks system analyseret, så skulle man kontakte matematikere 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 fungerer De fleste simuleringssoftware har den egenskab, at kunne animere model grafisk. Animationen er både brugbart for at kunne debugge modellen for at finde fejl, men også for at kunne demonstrere hvordan modellen fungerer. Animationen kan også hjælpe med at se hvordan systemet opfører sig i løbet af processen. Uden muligheden for animation, så ville simulation analyse havde været begrænset til mindre effektive tekstuelle og tal baseret præsentationer og dokumentation.

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.

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 parametre 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.

Dette afsnit vil beskrive de anvendte metoder til udarbejdelsen af denne rapport. Hvorfor disse metoder er blevet anvendt og hvilket formål de har haft vil blive beskrevet, samt hvilke tanker der har lagt til grund for dette igennem problemanalysen og problemløsningsafsnittet. Læseren kan med fordel se tilbage til dette afsnit, for at forstå hvordan der er blevet arbejdet igennem forløbet, skulle der opstå tvivl over hvordan vi er nået frem til vores deduktioner og resultater.

3.1 Problemområde

Problemområdet er det som danner grundlaget for projektet. Det har til formål at give et overblik over hvilke aspekter det valgte problem belyser, ud fra den initierende problem. Det klarelægger hvem problemet påvirker og hvem det vil gavne såfremt en løsning på problemet kan opnås til at analysere både problemets relevans og samt om det har nogle interessenter. Ud fra det initierende problem er der blevet stillet følgende hv-spørgsmål til at afgrænse problemets omfang;



Figure 3.1: Problemtræ over trafiksimulering

- Hvorfor: Hvad er årsagerne til problemet?
- Hvad: Hvem bliver ramt af problemet?
- Interessenter: Hvem er interessenterne - hvem har en interesse i en løsning på problemet?
- Hvor: Hvor findes problemet?
- Hvem: Hvem er hovedpersonerne i problemet?
- Hvordan: Hvordan kan problemet løses?

men også for at finde ud af at hvorvidt det opstillet problem har nogen relevans for at blive afviklet. Problemområdet er således blevet benyttet til at isolere alle de relevante aspekter af problemet, hvorefter det er blevet sat sammen i en større sammenhæng for at give et klart og tydeligt billede af hvor det præcise problem befinder sig til at udarbejde en problemformulering.

3.2 Fremgangsmåde

Under udarbejdelse af projektet er der blevet benyttet flere forskellige fremgangsmåder, alt efter hvad der er blevet arbejdet med under problemanalysen.

Fremgangsmåden under udarbejdelsen af rapporten er foregået ved at opstille relevante hypoteser, som enten kunne be- eller afkræftes ved at researche sig frem til eller ved hjælp af en prøve-fejl-metode.

3.2.1 Emperiske-induktive-metode

Under problemanalysen er den emperiske-induktive-metode blevet anvendt til at sikre os, at den information der er blevet indsamlet har statistisk belæg for deres udsagn eller er matematisk bevist. Dette gøre det muligt at udlede logiske slutninger til at underbygge rapportens argumentation, og sikre at troværdigheden i det som er blevet formidlet.

3.2.2 Andre metoder

Her tilføjes det hvis vi i løbet af rapporten anvender andre metoder.

3.2.3 Prøve-fejl-metoden

Under programmeringsfasen er der hovedsageligt blevet anvendt prøve-fejl-metoden, eftersom det har været den mest effektive metode til at opnå den ønskede effekt i programmet. Dette er blevet gjort ved at lave metoder, med et specifikt mål i tankerne til at udføre bestemte dele. Hvis metoden ikke opførte sig som forventede blev den omprogrammeret, indtil den bestemte metode udførte den ønskede effekt i programmet.

3.3 Kildekritik

For at finde frem til relevant information, er der blevet taget udgangspunkt i kilder med ophav fra statslige instanser eller anerkendte virksomheder og organisationer, for at sikre informationernes gyldighed og troværdighed. Dette vil det også være muligt at kunne kontakte kilderne for uddybende spørgsmål, skulle der opstå tvivl om dele af informationen eller indsamlet data i det anvendte information. Ydermere er der blevet forsøgt at finde den nyeste tilgængelige information, for at sikre at den indsamlede data stadigvæk er brugbart og gyldigt. Fordi flere af de anvendte kilder kunne have en tendens, er den præsenteret information blevet kritisk overvejet efter hvorvidt det forholder sig objektivt eller om det er blevet fremstillet til at opnå noget for egen vinding eller et specielt mål.

Den anvendte information er så vidt muligt også forsøgt at krydsrefereret med andre tilsvarende kilder, for at øge troværdigheden ved at undersøge om andre er nået frem til tilsvarende konklusioner og data.

Part I

Problemanalyse

Ændring af Konteksten

Som der beskrives i den initierende problemstilling, 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 tilfælde 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 sammenligning antallet af køretøjer med befolkningsantallet, vil man se en stigning i køretøjer per borger. I 1995 var der et køretøj til 41% af borgerne, og i 2016 er dette steget til et køretøj til 55% af borgerne. 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.

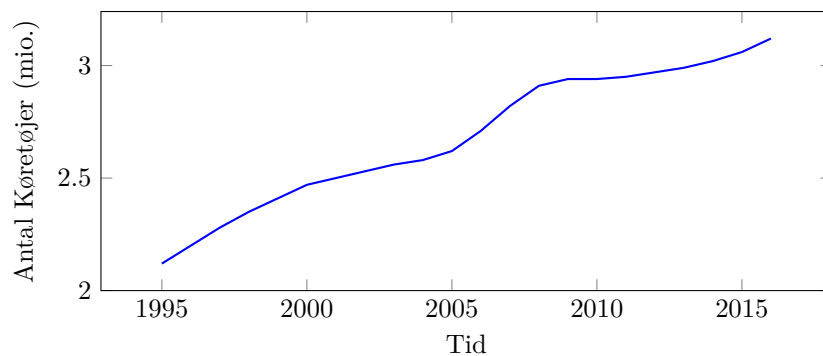


Figure 4.1: Væksten i bestanden af køretøjer

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 befolkningsantal. 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 essentielt 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 orientingsværktøjer 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.

Adfærdsmønstre er lige så uforudsigelige hvis ikke mere end nye teknologier. Heri består samfundsændringer der afgør landskabet som trafikmodellerne håndterer osv.

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 analyserer godstrafikken mellem Danmark og Tyskland, der er en meget avanceret model til trafikafviklingen i hovedstadsområdet, og en masse mindre regionale og kommunale modeller [14, 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 [14, s. 1]. Modeller på makroniveau danner et billede over den internationale situation [14, 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.(kilde speciale.)

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 trafikens bevægelsesmønster.(speciale) 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 [14, s. 9].

Mikroniveau På det mikroskopiske niveau er det kortsigtede modeller der bliver anvendt. Det område man undersøger er meget afgrænset. 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 [14, s. 1]. En af de anvendte modelleringsmetoder 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.(speciale)

Gennem problemanalysen vil en af de vedligeholdte modeller og en af de ikke vedligeholdte modeller blive undersøgt ved hjælp af en teknologianalysen, for at finde ud af hvilke elementer er vigtige. Informationen fra teknologianalysen kan derefter bruges til at lave en trafikmodel der er nem at vedligeholde.

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.

5.1 VisSim

VisSim er et mikrosimuleringsprogram, som bliver anendt 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.

5.1.1 Hvordan virker VisSim?

På figur 5.1 se at VisSim består af en værktøjslinje, som repræsenterer kommandoer og blokke. Disse blokke og diagrammer bruges til at forme simuleringen. Det er et blokprogrammerings sprog, man programmerer 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 [17].

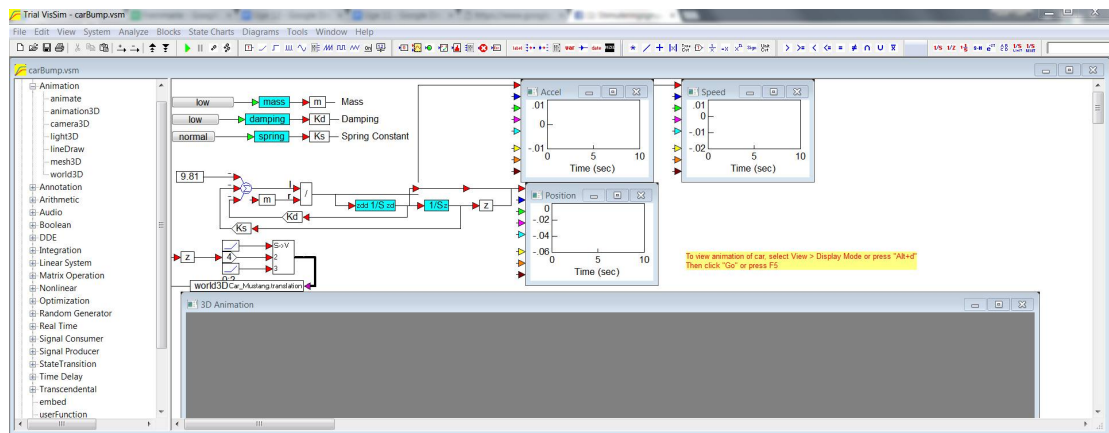


Figure 5.1: Værktøjer i VisSim

5.1.2 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 vurderer nogle af VisSims parametre, da det ikke er muligt for os at vurdere alle parametrene, fordi rapporten også fokuserer på andre aspekter end VisSim. Der vil også vurderes vedligeholdelsen af VisSim. Bilens parametre i VisSim er beskrevet nedenfor.

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

5.1.3 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.

5.1.4 Analyse af acceleration og deceleration

Ud fra en undersøgelse foretaget af Pihlkjær afgangsprøje 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 deceleration 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 i forkerte simuleringer, hvilket resultere i forkert planlægning af nye vejnetværk.

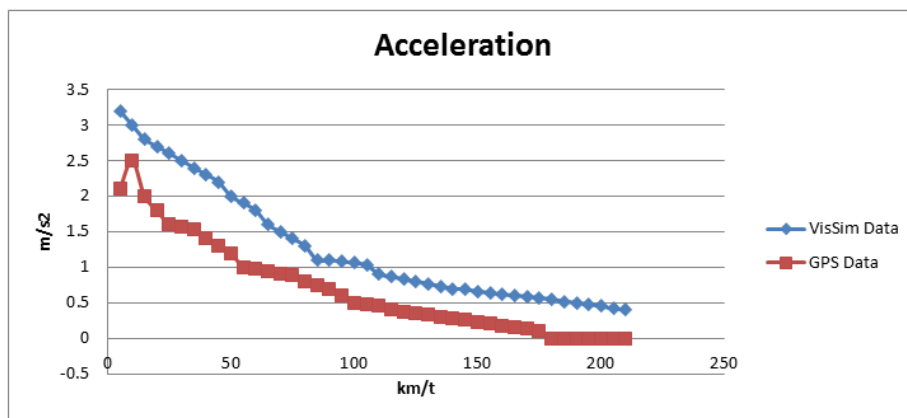


Figure 5.2: Visning af acceleration af VisSim og GPS data

Samtidig er der undersøgt af Pihlkjær afgangsprøje 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.

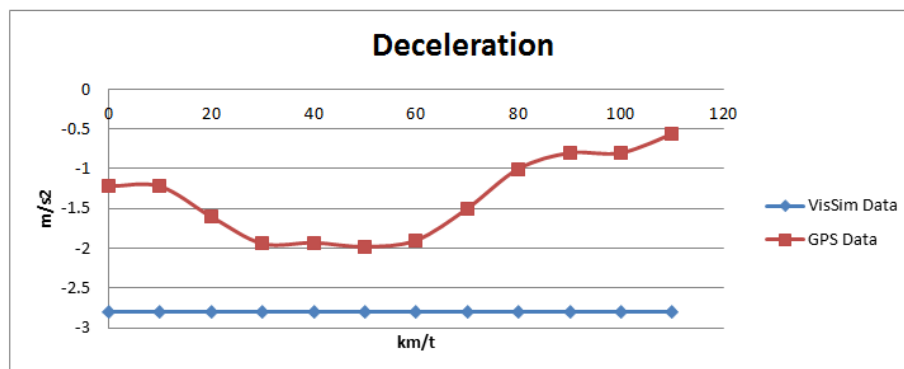


Figure 5.3: Visning af deceleration af VisSim og GPS data

5.1.5 Vedligeholdelse af VisSim

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 scenarier 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 stort 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.

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 [6, 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 [5, s. 14].

5.2.1 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 [6, 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 trafik-selskaber der bruger køreplanssystemet TR-System, DSB's data kommer i et andet format der bliver brugt til DSB's egen rejseplanlægger. For at beregne rejsetiderne af kørestrækningerne, samt ankomst og afgangstiderne sat op i et tredimensionalt 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 [6, s. 20-22].

Modellen af serviceniveau udregner serviceniveauet med variablerne tid, omkostninger og tilgængelighed. Modellen undlader at inkludere 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, afgangsfrekvenser 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 [5, 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 biler på vejnettet [6, s. 25]. Hastighederne modellen bruger til de forskellige veje, kan ses på tabel 5.1.

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

Table 5.1: Hastigheder på forskellige vejtyper

5.2.2 Adfærds model

Adfærds modellens formål er at give et estimat på fordelingen af transportmiddelvalg, populariteten af destinationer, kørekortfordeling, og bilejerskab. For at udregne estimaterne, benytter adfærds modellen sig af 3 undermodeller [6, 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 trafikken påvirkning på miljøet kan analyseres. Hovedformålet med at finde destinationerne er dog at modellere individer 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 [6, 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 [6, 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 [6, s. 29-30].

5.2.3 Emissions model

Man kan beregne emissioner af biltrafikken, dette gøres ved følgende: Der tages hensyn til og beregnes efter bilens tilstand, varm motor og koldstart. Det er en udregning der består ved at finde summen af en varstarts-emissionskoefficient gange trafikarbejdet og et koldstartstillæg for hver tur.

Disse faktorer er ikke selvstændige og bliver lavet per bilens årgang og dens størrelse/brændstoftype. Hastigheden af fartøjet determinere varmstarts-emissionskoefficienten. Der er sågar fortaget undersøgelser, såkaldte årskørsels undersøgelser af Vejdirektoratet hvorfra det er konkluderet at årskørsel forudsat er uafhængigt af bilens størrelse selvom at dette er set som urealistisk. Det estimeret trafikarbejde bliver udregnet ifølge en adfærdsmodel hvori man kigger på årskørsel pr bil i alle aldersgrupper. Der forøges eller reduceres med en faktor i selve fremskrivningsåret sådan at summen af antal biler i hver gruppe ganges deres gennemsnitlige årskørsel bliver lig det førnævnte trafikarbejde.

5.3 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 5.1. 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 modellørene muligvis have taget et andet avanceret 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 virkeligheden 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 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.

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 instancer 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 Transport- og Bygningsministeriets (TRM) - 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 [16]. 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 en gang 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.

Dette betyder at vi anser den kommunale sektor som en mulig interessant i den kontekst at informationen fra vores løsning ville kunne argumentere 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.

6.2.1 Sim Firmaer

Visual solution er et blandt andre firmaer der har lavet forskellige simuleringsværktøjer. VisSim er f.eks. et anerkendt værktøj som er anvendt af over 100.000 forskellige forskere verden rundt[3]. Vores simulerings program kan, med mere tid og udvikling potentielt blive en konkurrent til disse programmer og derfor menes det at disse firmaer er en modvirkende Interessant der kunne have en interesse i at modarbejde ideen.

Visual Solutions værktøj, VisSim, er en anerkendt standard for simuleringsværktøjer og deres værktøj, VisSim er derfor bl.a. Også blevet analyseret i denne rapportes 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 har før i tiden foretaget undersøgelse i sammenhæng med optimering af trængsel i trafik, miljøproblematikken og trafiksikkerhed[4]. I dette tilfælde ville værktøjet pivot[ordliste] mod en uddannelses kontekst hvilket på samme tid også kunne være en potentielt ide til videreudvikling. DTU er som sagt også ansvarlig for mange undersøgelser med anledning i trafik og kunne potentielt bidrage til udviklingen af softwareløsningen eller fremtidige iterationer af den.

Uddannelsessektoren tilbyder en interessant mulighed til at udarbejde projektet i en anden retning. Programmet skal i så fald være udarbejdet til at

passe til uddannelsesmiljøet hvilket vil sige at det skal kunne lære fra sig. Dette betyder at der skal laves undersøgelse i eksisterende uddannelses værktøjer.

6.4 Specificering af målgruppe

Ud fra interresantanalysen og teknologianalysen kan der delkonkluderes hvilken interessant der har størst interesse for vores projekt. Denne interessant er vurderet til at være kommunen baseret på at kommunerne står for størstedelene af vejnetværket i Danmark [1]. Derudover er den kommunale sektor anset for at være den mest realistiske interessant for gruppen at fokusere på iogmed at vi vil være i stand til at evt. kontakte en eller flere kommuner for at høre om hvad nogle krav kunne være til sådan en simulerings, software løsning. 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.

Problemformulering

I problemanalysen er der blevet foretaget teknologianalyser med fokus på vedligeholdelse af de undersøgte værktøjer, ydermere en analyse af forskellige interressanter har givet henblik på omfanget af løsningsmodellen. I teknologianalysen ses det at VisSim er ikke specifikt til trafiksimulering og er vanskeligt at opsætte for nyere brugere, derimod har Altrans ikke nok forskellige input til at kunne anvendes i flere kontekster. Samtidig er det blevet observeret, at VisSim er fleksibelt og tillader brugerne at specificere deres simulationer med deres egne parametre.

Altrans er modsat VisSim, opbygget til at finde fordelling af individer mellem den kollektive trafik og bilrejser, et specifikt formål. Der ses en mulighed for at lave en softwareløsning der har fleksibiliteten i stil med VisSim, men kun inde for et bestemt område, meso-trafik-simulering. Vores specificerede interessant, den kommunale sektor, lægger meget op til at opbygge netop et program med et meso omfang da der kan opsættes simulering af forskellige scenarier i netop dette mesosimulering. Vi formulere derfor følgende 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. En ideal løsning er at udarbejde et mesosimuleringsværktøj som kan tilpasse sig efter brugerens behov i den givne kontekst.

7.1 Ny

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 vedligeholdelse, trods ændringerne i konteksten?

Part II

Problemløsning

Til udformning af vores løsningsmodel bidrager dette afsnit en gennemgang af to algoritmer til ruteplanlægning og vejvisning. De er valgt på baggrund af deres udbredthed i industrien (KILDE). TEKNOLOGIANALYSEN anleder til at afgøre hvilke(n) algoritme(r) der er nødvendige til vores løsningsmodel. Der er blevet valgt to algoritmer til vejvisning som er A* og Dijkstras Algoritme. Der vil først blive gennemgået deres generelle funktion og dernæst en sammenligning af de to og en videre afgrænsning til hvilken der vil blive benyttet til løsningsmodellen.

8.1 Tidskompleksitet

Ved brug af tidskompleksitet kan man effektivisere tiden der bliver brugt af en computer til, at løse et problem ved brug af en algoritme. Det er en analyse af hvor lang tid det tager, at løse et problem med en bestemt størrelse. Tidskompleksiteten af en algoritme kan blive udtrykt i en bestemt mængde operationer, det tager for algoritmen at udføre. kompleksiteten kan være beregninger med heltal, dvs. addition, division, multiplikation og andre operationer. Tidskompleksitet er antallet af operationer der kræves i algoritmen og ikke computer tiden det tager at udføre algoritmen. Dette er pga. forskelling i tid det tager for forskellige computere at udføre disse grundlæggende operationer.[13]

8.1.1 Tidskompleksitet af iterative kontrolstruktur

På nedenstående pseudo kode er der en funktion A(), denne funktion indeholder en neded for løkke. Først initialiseret to int variable i og j, derefter bruges de i for løkken til at tælle op med. i den indre for løkke er der et funktions kald til en print funktion, som udskriver "test". For hver gang i tælles op til n gange, så tælles j også op n gange i den indre løkke. Dvs at print ("test") bliver udført n^2 gange. Dette betyder også at kompleksiteten er $O(n^2)$.


```

1  A()
2  {
3      int i, int j
4      for (i = 1 to n)
5          for (j = 1 to n)
6              print("test")
7  }

```

Figure 8.1: Tidskompleksitet iterativ

8.2 Grafteori

Grafteori handler om at beskrive modeller matematisk. Grafteori er væslig når man skal optimere et netværk, og derfor bruges det bla inden for rutefinding, hvor grafen kunne f.eks. representere et vejnetværk. En graf beskriver et par af mængder, man kunne tag udgangspunkt i grafen $G = (V, E)$ hvor V og E er mængder. I eksemplet her er V en ikke-tom mængde af knuder, og E er mængden af kanter som forbinder knuderne i mængden V (En kant kan godt forbinde den samme knude, dette kaldes en løkke).

Vi tager udgangspunkt i grafen.

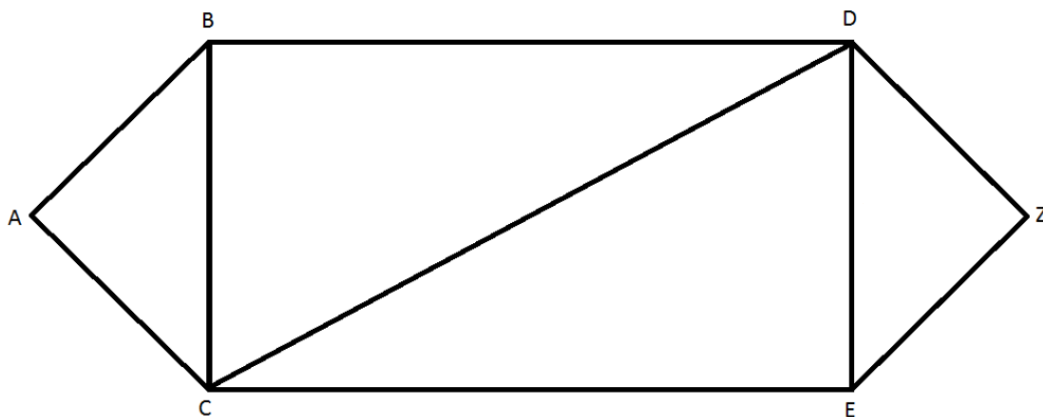


Figure 8.2: Et simpel vejnetværk

Man kan forstille sig dette er et vejnetværk, hvor punkterne(knuderne) repræsenterer byer, og linjerne(kanterne) er veje som forbinder byerne.

Vi kan udfra grafen se at knuderne $A, B, C, D, E, Z \in V$!. Samt at kanterne " $\{A,B\}$, $\{A,C\}$, $\{B,C\}$, $\{B,D\}$, $\{C,E\}$, $\{C,D\}$, $\{D,E\}$, $\{D,Z\}$, $\{E,Z\} \in E$ "

Grafen her kan derfor skrives rent matematisk:

$G = (V, E)$ og $V = \{A, B, C, D, E, Z\}$ og $E = \{ \{A,B\} , \{A,C\} , \{B,C\} , \{B,D\} , \{C,E\} , \{C,D\} ,$

En anden måde at repræsentere grafen på er ved hjælp af en adjacency

matrix $V \times E$, hvor $v_1, v_2 \dots v_n$, er knuderne, og $e_1, e_2 \dots e_n$, er kanterne. I matrixen beskrives forbindelser mellem 2 knuder med 1 taller, hvor 0 beskriver ingen forbindelse mellem de 2 knuder.

$$G = \begin{matrix} & \begin{matrix} A & B & C & D & E & Z \end{matrix} \\ \begin{matrix} A \\ B \\ C \\ D \\ E \\ Z \end{matrix} & \begin{bmatrix} 0 & 1 & 1 & 0 & 0 & 0 \\ 1 & 0 & 1 & 1 & 0 & 0 \\ 1 & 1 & 0 & 1 & 1 & 0 \\ 0 & 1 & 1 & 0 & 1 & 1 \\ 0 & 0 & 1 & 1 & 0 & 1 \\ 0 & 0 & 0 & 1 & 1 & 0 \end{bmatrix} \end{matrix}$$

Figure 8.3: Adjacency matrix

Grafteori er derfor 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 enkelte knude, for at kun udregne den korteste rute. Derudover vil grafmetoder som nabo-matrixen være en god måde at beskrive en grafs mængder i programmerings niveau.

8.3 Dijkstras Algoritme

Dijkstras algoritme er en algoritme til at finde den korteste vej fra et bestemt punkt til et andet punkt. Disse punkter kan bland andet repræsentere den korteste vej mellem to forskellige byer. Dijkstra fungerer på nogenlunde samme måde som A^* i dens planlæggende fremgangsmåde ved at sammenligne punkter med hinanden. Dijkstras algoritme skanner også et område af kasser fra et startpunkt og fortsætter som eksemplificeret i figur 8.6. Dijkstra modtager dog ikke heuristiske datasæt Dijkstras er en grådig algoritme, da den finder den korteste længde først og fortsætter således.[12] Dette kaldes også Greedy Best First Search.

Dijkstras algoritme finder den korteste rute mellem 2 forskellige punkter i en simpel ikke-orienteret vægtet graf. Man kan se på figur 8.4 at der angives forskellige punkter A, B, C, D, E, Z. Hvis man skal fra punkt A til Z på figuren, så starter man ved A og derfor initialiseres A til at være 0, som man kan se på tabel 8.1. Algoritmen virker således, at alle punkter er uendeligt udover det punkt man befinder sig på som vist på tabel 8.1. Algoritmen tager punkt fra punkt, så den starter med at se de grene som A har. Disse er $|AB| = 4$ og $|AC| = 2$. Her fra kan algoritmen ikke se videre end B og C. Den ser altid på det mindste tal, og derfor tager den længden fra A til B som er 4 og længden fra A til C som er 2, begge disse tal er mindre end uendeligt. Herefter ser den efter hvilket af de nuværende tal som er mindst, hvilket er 2. Så derfor vælger den C som sit næste punkt. Algoritmen finder nu de næst tætteste punkt, ved at addere alle de tidligere ruter, som har den korteste rute fra A til det næste sæt af punkter. Her ser algoritmen ud fra C og hvilke grene C har. Dette er længden til B, D og E, dog er længden altid fra A, så derfor er længden fra A til E 12 da $2+10 = 12$. Længden til B er nu blevet 3, da algoritmen ser på den korteste rute, så A til B er 3, da $2+1 = 3$. Således fortsætter algoritmen indtil den rammer Z.[12]

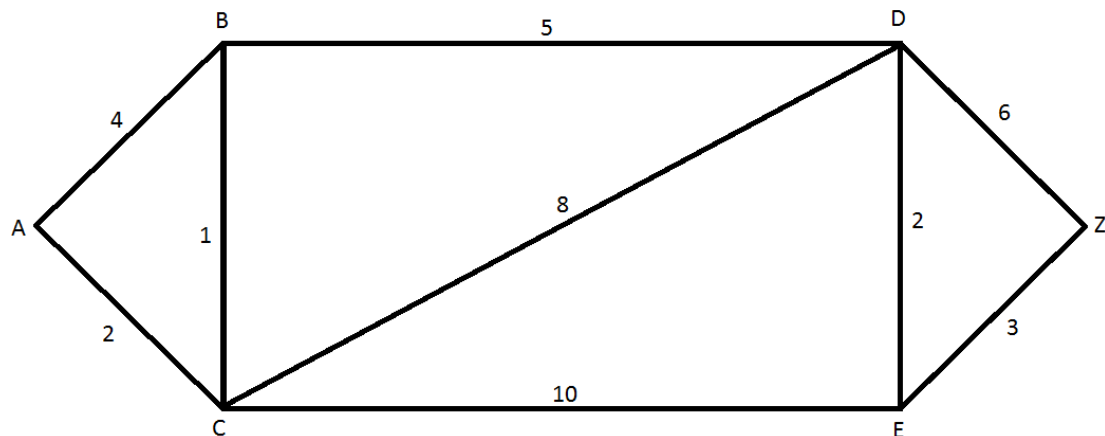


Figure 8.4: Graf til fremvisning af eksempel af Dijkstras algoritme i brug

	A	B	C	D	E	Z
	0	Inf	Inf	Inf	Inf	Inf
A	0	4	2	Inf	Inf	Inf
A, C	0	3	2	10	12	Inf
A, C, B	0	3	2	8	12	Inf
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 tabel

```

1  procedure Dijkstras(G: weighted connected simple graph, with  $\leftarrow$ 
   all weights positive)
2  {G has vertices a = V0, V1, ..... Vn = z and lengths w(Vi, Vj)  $\leftarrow$ 
   where w(Vi,Vj) = infinity if {Vi, Vj} is not an edge in G}
3  for (i = 1 to n)
4    L(Vi) = infinity
5  L(a) = 0
6  S = NULL
7  { the labels are now initialized so that the label of a is 0 and  $\leftarrow$ 
   all other labels are infinity, and S is the empty set }
8  while (z does not belong to S)
9    u = a vertex not in S with L(u) minimal
10   S = S U {u}
11   for (all vertices v not in S)
12     if (L(u) + w(u, v) < L(v) then L(v) = L(u) + L(u, v))
13     {this adds a vertex to S with minimal label and updates the  $\leftarrow$ 
   labels of vertices not in S}
14  return (L(z)) {L(z) = length of a shortest path from a to z}

```

Listing 8.1: Dijkstras angivet som eksempel i pseudo-kode

8.4 A* Algoritmen

Primært når det kommer til belægning af en dynamisk rute, foregår det ved at en enhed fortsætter hen i mod et mål indtil den når en forhindring. Dette er et ekstremt simpelt bevægelsesmønster og indebærer in vis in-effektivitet. Rent retorisk kunne man stille spørgsmålet om det ikke ville være smartere at planlægge en rute før man overhovedet bevæger sig.

A* er en algoritme til at beregne den korteste rute baseret på en række heuristiske datasæt. A* får input igennem en brugerlavet graf der indeholder en række datasæt for at algoritmen kan fungere. Først har vi distancen fra punkt til punkt, eksempelvis punkt 'A' til punkt 'B' som vi kalder for f.eks. **H** og dernæst har vi et datasæt **G** der indeholder bekostningen for at flytte fra en kant til en

anden, denne variabel er bestemt på forhånd. Et virkelighedseksempel kunne være at man vil over på den anden side af en sø, så har man så muligheden for at svømme direkte eller gå uden om og det koster f.eks. 2 gange så meget at bevæge sig direkte igennem søen. Dette er givet ved G , hvor som sagt H er den ultimative korteste længde til det bestemte slutpunkt. H fungerer desuden for hvilket som helst punkt i et system og angiver *altid* den korteste vej til slutpunktet uanset forhindringer. Det skal også nævnes at H ikke er påvirket af bevægelsesbekostningen, til at starte med, som G angiver, dette kommer først senere. Til sidst har vi F der er en sammenlagt værdi af både H og G . Dette gælder kun for hver kasse der flyttes til, hvori H er angivet ved kassen man flytter tils H værdi. Det kan vises således i formelen 8.1:

$$F(n) = G(n) + H(n) \quad (8.1)$$

En måde man kan visualisere A^* på er f.eks. med et gitter-system som set i figur 8.5. Her kan vi se at vi har et start punkt (grøn) og et slutpunkts (blå). De kasser vi ikke kan bevæge os igennem er de røde kasser. Figuren angiver ingen heuristiske datasæt endnu.

1	2	3	4	5	6	7	8	9	10	11	12
13	14	15	16	17	18	19	20	21	22	23	24
25	26	27	28	29	30	31	32	33	34	35	36
37	38	39	40	41	42	43	44	45	46	47	48
49	50	51	52	53	54	55	56	57	58	59	60
61	62	63	64	65	66	67	68	69	70	71	72
73	74	75	76	77	78	79	80	81	82	83	84
85	86	87	88	89	90	91	92	93	94	95	96

Figure 8.5: A^* gitter-system

Ud fra figuren kan vi begynde os at forestille hvordan A^* fungerer. Når man bevæger sig fra kasse til kasse laver man 2 lister til at holde styr på hvor brikken har været. En liste til at holde styr på hvilke kasser man ikke har besøgt endnu og en liste der holder styr på hvilke man **har** besøgt. Når man flytter brikken

skal man derfor angive hvilken kasse der nu skal på *besøgt* listen. Derfor som nævnt skal vi bruge information om hvor meget **G** koster. Brikken skal nu til at flytte sig for at komme til slutpunktet. Dette kunne f.eks. være 10 point for at flytte sig i hvilken som helst retning, men man kunne også sagtens angive at diagonal bevægelse ville koste 12 point. Dvs. at ruten ændrer sig til måske ikke at være så direkte som den ellers kunne have været.

Der findes flere metoder man kan anvende A* på og en af dem vises her. Det vises her i den lille bid af Python-kode i Listing 8.2:

```
1 frontier = Queue()
2 frontier.put(start)
3 visited = {}
4 visited[start] = True
5
6 while not frontier.empty():
7     current = frontier.get()
8     for next in graph.neighbors(current):
9         if next not in visited:
10             frontier.put(next)
11             visited[next] = True
```

Listing 8.2: A stjerne og pseudo-kode af brug af lister

[11]

Som set i figur 8.5 har vi vores liste givet ved kassernes nummerering. Nummereringen kører fra venstre mod højre én række ad gangen. Vi angiver at det tager 10 point at gå lodret og vandret én kasse ad gangen og 12 point at gå diagonalt. I figur 8.6 kan vi nu se de heuristiske datasæt angivet fra startpunktet (grøn). Hver enkel kasse omkringliggende startpunktet har deres **H** værdi angivet med lys-lilla tekst og bevægelsesomkostningen **G** fra startpunktet til kassen angivet i blå tekst.

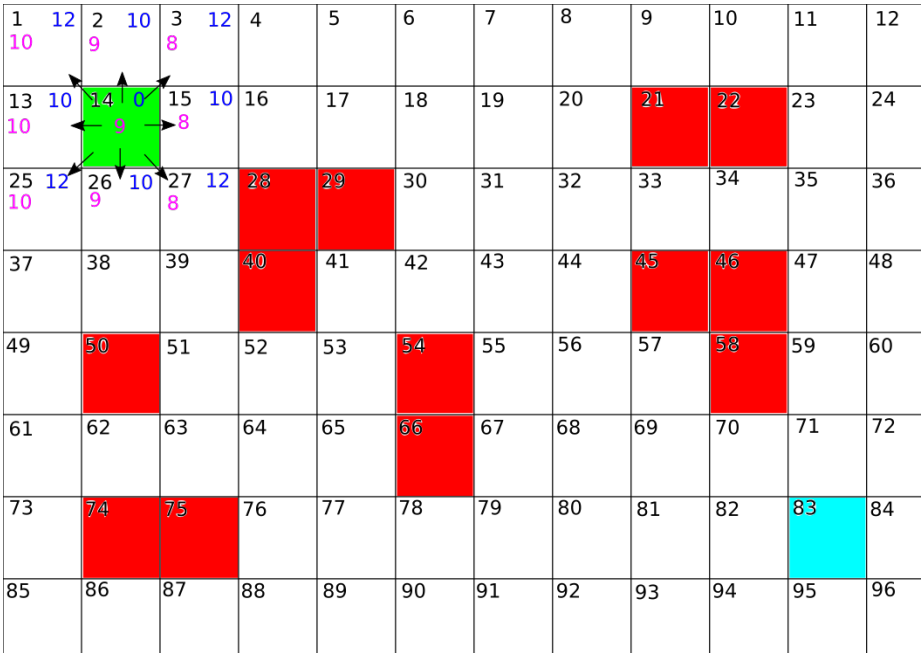


Figure 8.6: A* der viser bekostning af bevægelse fra startpunkt (grøn) til omkringliggende kasser (G) angivet med blå farve samt H angivet med lys-lilla

Nu udregnes \mathbf{F} værdien så f.eks. hvis vi går fra kasse 14 (startpunktet) til 15 skal vi lægge 10 (\mathbf{G}) og 8 (\mathbf{H}) sammen. Dette gør vi så for alle omkringliggende kasser for startpunktet. Dernæst går man til den laveste \mathbf{F} værdi og gør helt det samme som før, derudover flyttes den nye kasse man står på til *besøgt* listen. Noget man skal være opmærksom på her er at man stadig skal sammenligne bevægelsesomkostningen fra den tidligere kasse til de kasser der også er relevante for den nye kasse man har flyttet sig til. For dermed at afgøre om man kunne have påført en smartere bevægelse. Det skal igen pointeres at \mathbf{A}^* modtager data fra en graf og det her modelleres.

Denne fremgangsmåde er også bedre kendt som Breadth First Search, hvori en frontlinje bliver kontinuerligt fremskyndet baseret på omkostninger og heuristiske datasæt[11]. A^* er en heuristisk fremgangsmåde afledt af Dijkstras generelle funktionalitet. Dijkstra og A^* vil altid give en kortest vej hen til målet.

Kravspecifikationer

1. Generelt

Gem og Åben For at brugeren ikke skal starte forfra ved genstart af programmet, skal der være mulighed for at gemme vejnettet i en fil, og åbne det igen på et senere tidspunkt.

Importering fra kort Indlæs fra OpenStreetMap og opstil dataen for et område i programmet.

2. Brugerflade

Gitter Ved indsætning af veje og elementer som lyskryds, vil man indsætte disse i et gitter system, der kan panoreres, forstørres og formindskes. Formålet med gitteret er at kunne opstille vejnettet på en systematisk måde.

Knudepunkter Bruges til at forbinde veje.

Veje Tegnes mellem knudepunkterne, og har en maksimal hastighed.

Eksklusive Veje Ligesom almindelige veje, de bliver kun taget med i en af simuleringerne, så der kan måles på forskellen disse veje gør for trafikken.

Kryds Man skal kunne indsætte komplekse elementer som eksempelvis kryds med signaler.

Vigepligt Brugeren skal være i stand til at specificere hvilke veje, der har vigepligt i et kryds.

Huse Huse indsættes og bruges som bilrejsernes startpunkt, og slutpunkt ved tilbage rejsen.

Destinationer Brugeren skal kunne opstille forskellige typer destinationer og indsætte disse i gitteret. Ved simuleringen af vejnettet, vil brugeren så være i stand til at bestemme hvor stor en procentdel af bilisterne der vælger en destination.

Parkeringspladser For at billisterne ikke bare parkerer på destinationen, vil der skulle indsættes parkeringspladser. Disse parkeringspladser vil være slutpunktet for rejsen fra billistens hus, hvor billisten søger efter den parkeringsplads der er nærmest destinationen.

Togbaner Simuleringen skal tage højde for togenes påvirkning på trafikken.

Fodgængerfelt Trafikkens hastighed bliver påvirket af fodgængere.

3. Simulering

Sammenligning For at simulering kan bruges til at vurdere et system, skal man være i stand til at sammenligne resultaterne med en anden simulation. Dette vil foregå ved at brugeren kan opstille særlige veje, der kun bliver medtaget i en af simulationerne, og brugeren vil derefter kunne sammenligne outputtet.

Fodgængere Simulering af fodgængerne med formålet at gøre simuleringen mere virkelighedsnær, i det at bilerne er nødt til at vente når fodgængerne passerer vejen.

Cyklister Lige som fodgængere påvirke cyklister også trafikken i nogle kryds.

Køretøjer Det skal være muligt at opstille forskellige typer køretøjer som for eksempel biler og busser.

Pathfinding Køretøjerne skal kunne beregne den kortest mulige rute til destination. Ved denne beregning skal der overvejes fartgrænser på vejene, derudover skal der tages højde for elementer som lyskryds.

Acceleration Køretøjerne skal ikke kunne nå hastighedsgrænsen øjeblikkeligt.

Deceleration Køretøjerne skal ikke kunne bremse øjeblikkeligt.

(a) Input

- i. Antallet af køretøjer
- ii. Gennemsnitshastighed for køretøjer
- iii. Destinationsvalg

(b) Output

- i. Procent af rejsetid brugt i kø
- ii. Procent af rejsetid brugt i en hastighed lavere end den ønskede hastighed.

9.0.1 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 simuleringsenheder 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 i perspektiverings afsnittet (reference her).

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 bevægelse 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.

9.1 Black Box Test - Unit Test

Black Box Test

Black box testing er en software metode som man benytter til at teste sit program for interne fejl. Denne metode er den måde de fleste tester deres program, og bliver brugt i størstedelen af det praktiske liv.

Denne metode er kaldt "Black Box", da man behandler den som en sort box, hvor man ikke kender det interne struktur af den software man vil teste. Når så man vil teste programmet, så gør man det fra kunders synspunkt. Når så man tester programmet, så kender kunden ikke til hvordan programmet virker, så han sidder med noget input som han vil putte ind i denne "Black Box", hvor outputtet er noget man forventer der har været igennem en proces i programmet og udarbejdet til noget man kan bruge[1]. Det vigtigste formål med dette er at kontrollere om softwaret arbejder som der forventes af det, altså det opfylder kravene som der er blevet sat og det opfylder kunders forventninger til det. Der er forskellige måder at teste der bliver benyttet i industrien. Hver testing måde har deres egen fordele og ulemper.

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 individuel del af programmet virker og om de er egnede til brug. Unit tests er skrevet og benyttet af programmører, for at være sikre 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 fungerer 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[2].

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.

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 det rette fremgangsmåde er 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.

10

Implementation

I dette kapitel vil vi beskrive implementeringen af de forskellige dele af løsningsforslaget.

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. De følgende diagrammer er de endelige udgaver, under udformningen af programmet er der blevet ændret, tilføjet og fjernet forskellige klasser som diskuteres i afsnit **REF**. Klassediagrammerne for brugerfladen er ikke vist her, men kan findes i bilag **REF**.

I diagrammerne betyder skråtekst at klassen er abstrakt, plus er et offentligt medlem, minus er et privat medlem, hashcode er et beskyttet (protected) 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.

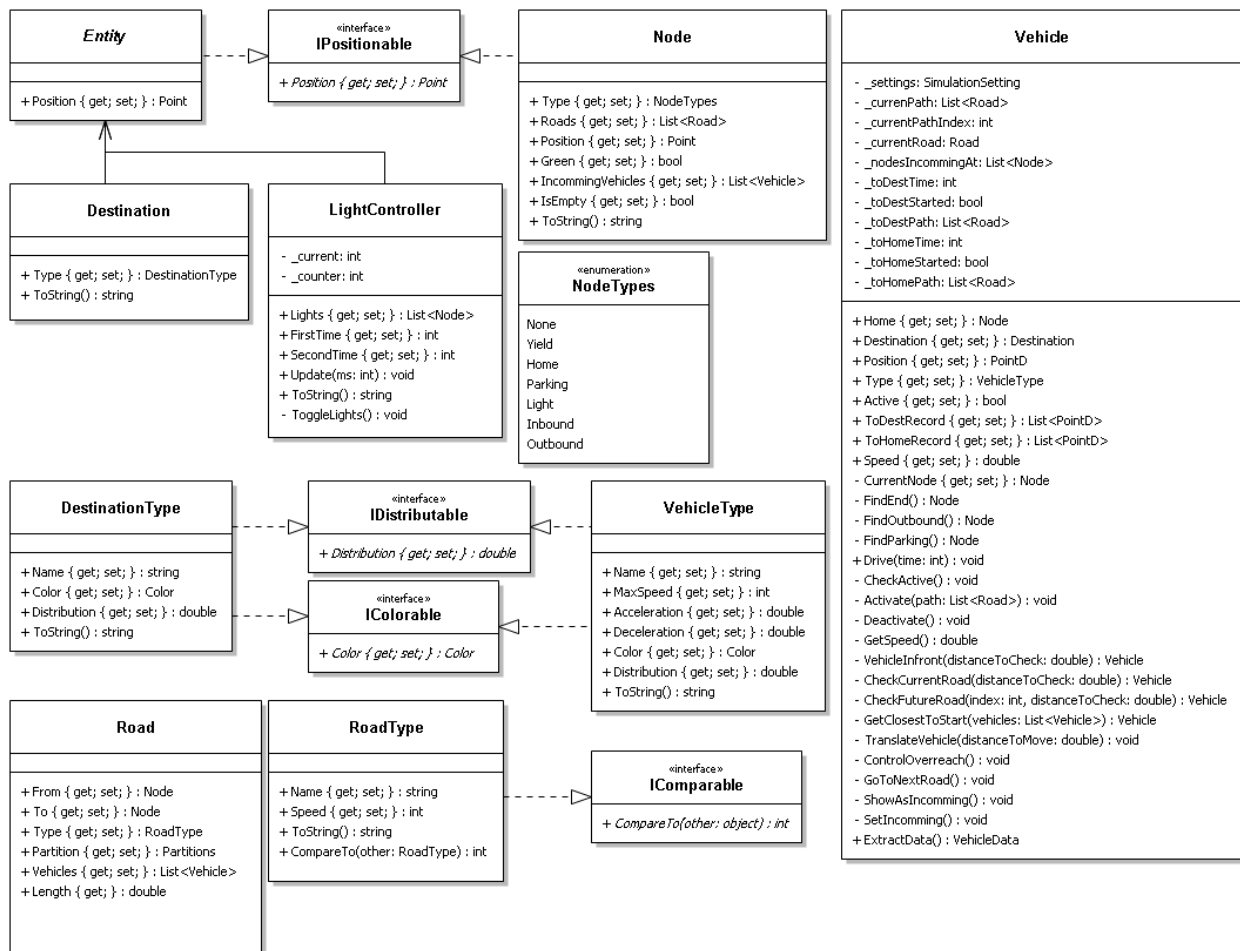


Figure 10.1: Elementer i vejnettet

Diagrammet der ses på figur 10.1 indeholder alle klasserne der er en del af vejnettet. Elementerne der arver fra den abstrakte klasse **Entity**, er elementer der kan positioneres, men ikke forbindes med vejnettet. **Node** klassen er de noder vejne kan forbindes imellem. **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 **REF**, bortset fra **Vehicle** der bliver forklaret i afsnit **REF**.

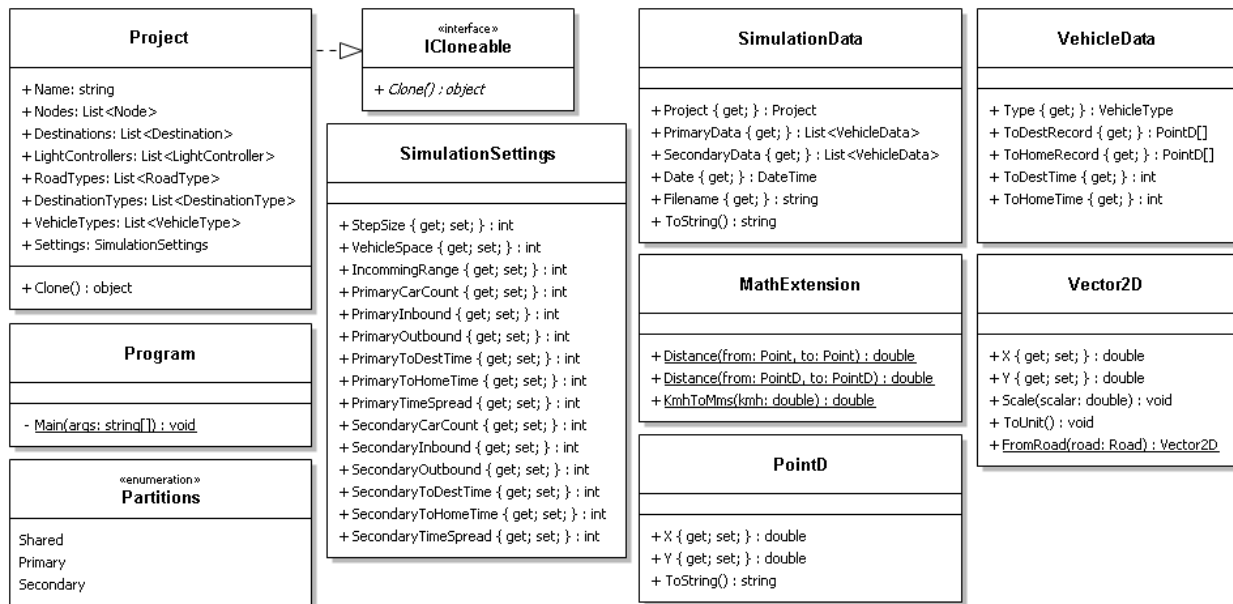


Figure 10.2: Diverse klasser

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 **REF**.

Viewport	SimulationViewPort
<ul style="list-style-type: none"> + GridLength: int = 1000 + GridSize: int = 16 + EntitySize: int = 12 + NodeSize: int = 8 + Project: Project + HoverConnection: Point + MousePos: Point - _viewPos: Point + Grid: Layer + Connections: Layer + Nodes: Layer + Entities: Layer + Information: Layer + Input: Layer 	<ul style="list-style-type: none"> - VehicleSize: int = 16 + Vehicles: Layer + SimData: SimulationData + CurrentPartition: Partitions
<ul style="list-style-type: none"> + GridPos { get; } : Point + ViewPos { get; } : Point + Zoom { get; set; } : float + Reset() : void - SetZoom(value: float) : void - SetViewPos(value: Point) : void - GetGridPos() : Point - OnMove(sender: object, args: MouseEventArgs) : void - OnWheel(sender: object, args: MouseEventArgs) : void + GetObjByGridPos() : object - InitControls() : void - GetDrawPosition(position: Point) : Point # ScaleTranslateSmooth(mode: SmoothingMode, args: PaintEventArgs) : void # DrawGrid(sender: object, args: PaintEventArgs) : void - DrawConnections(sender: object, args: PaintEventArgs) : void - DrawRoad(road: Road, args: PaintEventArgs) : void - DrawNodes(sender: object, args: PaintEventArgs) : void - DrawNode(fill: Brush, position: Point, args: PaintEventArgs) : void - DrawArrow(node: Node, left: bool, args: PaintEventArgs) : void - DrawEntities(sender: object, args: PaintEventArgs) : void - DrawLightController(position: Point, args: PaintEventArgs) : void - DrawDestination(color: Color, position: Point, args: PaintEventArgs) : void - DrawInformation(sender: object, args: PaintEventArgs) : void 	<ul style="list-style-type: none"> + Time { get; set; } : int - DrawVehicles(sender: object, args: PaintEventArgs) : void - DrawVehicle(vehicle: VehicleData, toDest: bool, args: PaintEventArgs) : void - GetDrawPosition(position: PointD) : PointF - GetRecordIndex(recordStartTime: int) : int

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 **REF**.

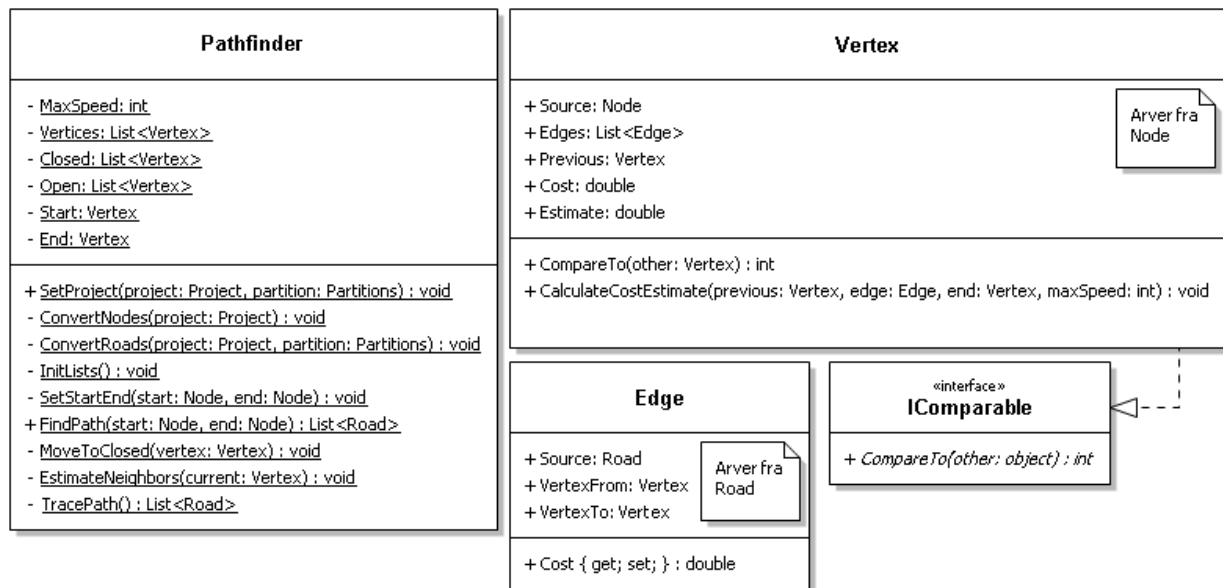


Figure 10.4: Pathfinder klassen

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 **REF**.

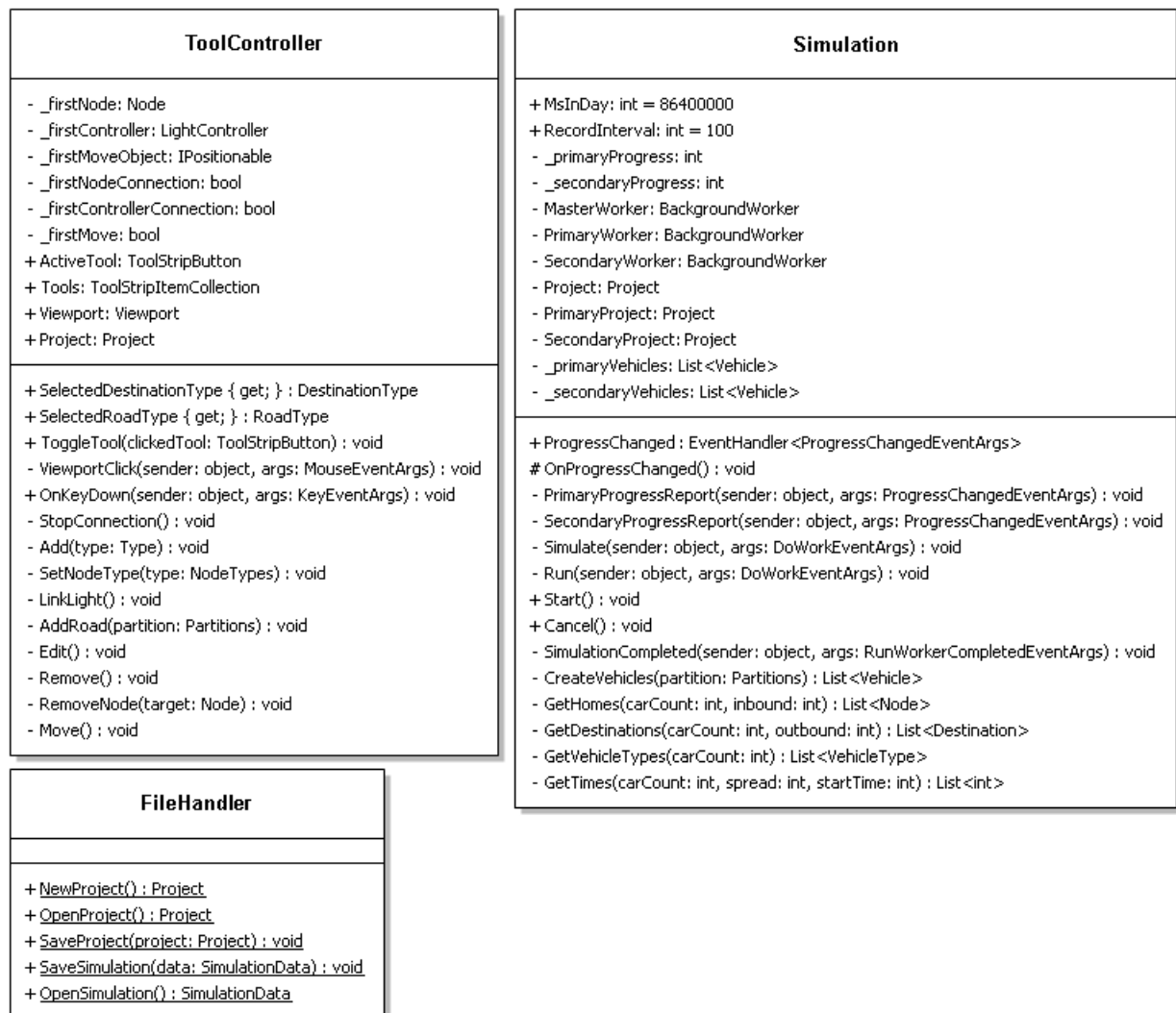


Figure 10.5: De funktionelle klasser

Diagrammet på figur 10.5 viser klasserne hvor en stor del af programmets logik bliver håndteret. **ToolController**erne modtager input fra **GUIMain** når der bliver trykket på værktøjsknapperne, og modtager input fra **Viewport**en 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 **REF**.

10.2 Hoved Brugerfladen

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

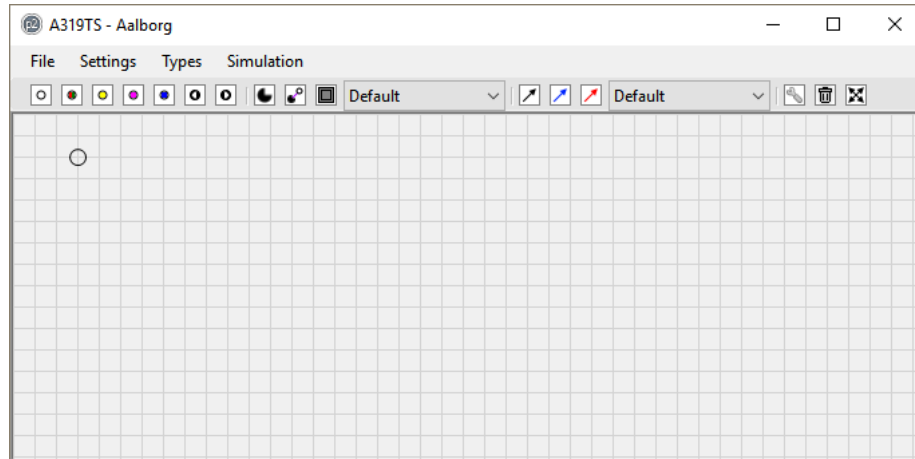


Figure 10.6: Programmets hoved vindue **GUIMain**

På figur 10.6 ses **GUIMain**. **GUIMain** består af en menulinje, værktøjslinje og en Viewport. Vinduet fungerer udelukkende ved brug af events. 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 håndterer 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** der bliver brugt, og i listen til højre kan brugeren vælge hvilken **RoadType** 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

10.3.1 Node

I vores program anvender vi et grid hvorpå brugeren indsætter noder der udgør de forskellige vejnet der bliver oprettet. Disse noder kan være forskellige typer. Dette er gjort således at processen i at oprette vejnet er relativt simple. Et simpelt eksempel ville være at brugeren indsætter to noder, den første node hvor brugeren ønsker køretøjerne skal køre fra, og en node 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.

10.3.2 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 **DestinationType** klassen, dette er, ligesom med **Vehicle** og **Road** klasserne, en klasse der bruges til at brugerdefinere forskellige typer af destinations med forskellige parametre.

10.3.3 LightController

LightController klassen er den del af programmet hvori brugeren kan indstille på deres trafiklys noders 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 10.7: LightController klassen

LightControllerens funktioner er at skifte trafiklys fra rød til grøn og tilbage igen. Dette er muligt at gøre ved to separate intervaller sådan at trafiklysene skifter efter forskellige mængder tid. Dette valg blev truffet efter det blev iagtaget at denne samme opførsel ses også på trafikryds rundt omkring i Aalborg. Dette er idelt hvis brugeren prøver at simulere et trafikryds hvor den

ene vej er mere anvendt en den anden, f.eks. En vej der leder ind i byen fra en motorvej der krydser med en vej der leder ind til mindre boligkompleks.

10.3.4 Road

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

```
1 [Serializable]
2 public class Road
3 {
4     public Node From { get; set; }
5     public Node To { get; set; }
6     public RoadType Type { get; set; }
7     public Partitions Partition { get; set; }
8     public List<Vehicle> Vehicles { get; set; }
9     public double Length
10    {
11        get { return MathExtension.Distance(From.Position,
12                                             To.Position); }
13    }
14    public Road(Node from, Node to, RoadType type,
15               Partitions partition)
16    {
17        From = from;
18        To = to;
19        Type = type;
20        Partition = partition;
21        Vehicles = new List<Vehicle>();
22    }
23 }
```

Figure 10.8: Road klassen

Road klassen tager imod en instans af **RoadType** som brugeren selv har defineret via **RoadType** klassen. På denne måde har brugeren kontrol over hvilken type vej der tale om og hvordan vejen opfører sig i programmet, f.eks. om det er en motorvej. Vejene er programmeret således at brugeren selv kan opstille forskellige kryds, rundkørsler eller andre avanceret afkørsels baner. **RoadType** er en separat klasse hvori brugeren give et **Name** og en **Speed**, på denne måde definere brugeren selv hvilke typer veje der optræder i deres simulation.

10.3.5 Typer

10.4 Diverse

10.5 Project

Project klassen har til formål at sætte de forskellige types klasser op i lister, mens Project (string name) samtidig sætter default values når man laver en ny RoadType, DestinationType eller VehicleType i programmet, hver gang der bliver "addet" noget nyt til de forskellige lister under Types. Project Clones gør det muligt at gemme det man har lavet i en ny fil, som så kan bruges

senere såfremt det skal sættes ind andre steder i programmet ved eksempelvis at copy-paste det.

```

1
2
3 {
4     [Serializable]
5     public class Project : ICloneable
6     {
7         public string Name;
8         public List<Node> Nodes = new List<Node>();
9         public List<Destination> Destinations = new List<Destination>();
10        public List<LightController> LightControllers = new List<LightController>();
11        public List<RoadType> RoadTypes = new List<RoadType>();
12        public List<DestinationType> DestinationTypes = new List<DestinationType>();
13        public List<VehicleType> VehicleTypes = new List<VehicleType>();
14        public List<SimulationData> Simulations = new List<SimulationData>();
15        public SimulationSettings Settings = new SimulationSettings();
16
17        public Project(string name)
18        {
19            Name = name;
20            RoadTypes.Add(new RoadType("Default", 50));
21            DestinationTypes.Add(new DestinationType("Default", Color.LightSlateGray) { Distribution = 100 });
22            VehicleTypes.Add(new VehicleType("Default", 130, 5, Color.LightSlateGray) { Distribution = 100 });
23        }
24
25        public object Clone()
26        {
27            MemoryStream memory = new MemoryStream();
28            BinaryFormatter formatter = new BinaryFormatter();
29            formatter.Serialize(memory, this);
30            memory.Position = 0;
31            return formatter.Deserialize(memory);
32        }
33    }

```

Figure 10.9: Project.cs

10.6 SimulationSettings

SimulationSettings sætter defaultværdierne til de forskellige variabler, og giver brugeren muligheden for at ændre dem og gemme dem med nye værdier i stedet for de predefineret values, skulle man fortryde de selvdefineret values der er blevet sat, kan man altid vælge SetDefault values, ændre ens settings tilbage til de originale default values.

```

1  {
2
3      [Serializable]
4      public class SimulationSettings
5      {
6          // Shared
7          public int StepSize { get; set; }
8          public int VehicleSpace { get; set; }
9          public int IncommingRange { get; set; }
10
11         // Primary
12         public int PrimaryCarCount { get; set; }
13         public int PrimaryInbound { get; set; }
14         public int PrimaryOutbound { get; set; }
15         public int PrimaryToDestTime { get; set; }
16         public int PrimaryToHomeTime { get; set; }
17         public int PrimaryTimeSpread { get; set; }
18
19         // Secondary
20         public int SecondaryCarCount { get; set; }
21         public int SecondaryInbound { get; set; }
22         public int SecondaryOutbound { get; set; }
23         public int SecondaryToDestTime { get; set; }
24         public int SecondaryToHomeTime { get; set; }
25         public int SecondaryTimeSpread { get; set; }
26
27         // Defaults
28         public SimulationSettings()
29         {
30             StepSize = 100;
31             VehicleSpace = 2;
32             IncommingRange = 10;
33             PrimaryCarCount = 1000;
34             PrimaryInbound = 100;
35             PrimaryOutbound = 100;
36             PrimaryToDestTime = 28800000; // 08:00
37             PrimaryToHomeTime = 57600000; // 16:00
38             PrimaryTimeSpread = 14400000; // 4h
39             SecondaryCarCount = 1000;
40             SecondaryInbound = 100;
41             SecondaryOutbound = 100;
42             SecondaryToDestTime = 28800000; // 08:00
43             SecondaryToHomeTime = 57600000; // 16:00
44             SecondaryTimeSpread = 14400000; // 4h
45         }
46     }
47 }

```

Figure 10.10: SimulationSettings

10.6.1 Data

Dataen fra simuleringen i programmet gives til brugeren igennem to klasser. SimulationData, er en klasse som modtager to lister fra VehicleData og derudover tager den imod information fra Project og Datetime. Metoden SimulationData læser disse data ind til de forskellige typer i klassen. Der returneres dernæst Project.name og en DateTime type.

```

1 public class SimulationData
2 {
3     public Project Project { get; private set; }
4     public List<VehicleData> PrimaryData { get; private set; }
5     public List<VehicleData> SecondaryData { get; private set; }
6     public DateTime Date { get; private set; }
7     public string Filename
8     {
9         get { return Project.Name + "_" + Date.ToString("dd") + "-" +
10             Date.ToString("MM") + "-" +
11             Date.ToString("yy") + "_" + Date.ToString("HH") + "-" +
12             Date.ToString("mm") + ".sim"; }
13     }
14     public SimulationData(Project project, List<VehicleData> primary, List<VehicleData> secondary)
15     {
16         Project = project;
17         PrimaryData = primary;
18         SecondaryData = secondary;
19         Date = DateTime.Now;
20     }
21     public override string ToString()
22     {
23         return Project.Name + " " + Date;
24     }
25 }

```

Figure 10.11: SimulationData Klasse

Som nævnt over figuren så bliver der anvendt en klasse til SimulationData. VehicleData er som pointeret data der bliver lagt over i to liste, PrimaryData og SecondaryData. VehicleData består ikke af nogle større funktioner og derfor vil der ikke være nogle dybbere forklaring dertil. Den består af nogle typer der læser data fra PointD klassen og VehicleType.

10.6.2 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 kmh til mms. Der gøres brug af klasser fra math library. Disse udregninger anvendes bl.a i Vector2D klassen.

10.6.3 Vector2D

Vector2D klassen anvendes til at opsætte de metoder som gør det muligt at bl.a. vehicles kan udregne hvor hen de skal bevæge sig i simuleringen. Metoderne er for det meste til at indlæse koordinater på griddet i simuleringen til anvendelse når simuleringen kører.

```
1 class Vector2D
2 {
3     public double X { get; set; }
4     public double Y { get; set; }
5     public double Length
6     {
7         get { return MathExtension.Distance(new PointD(0, 0), new ←
            PointD(X, Y)); }
8     }
9
10    public Vector2D()
11    {
12        X = 0;
13        Y = 0;
14    }
15    public Vector2D(double x, double y)
16    {
17        X = x;
18        Y = y;
19    }
20
21    public void Scale(double scalar)
22    {
23        X *= scalar;
24        Y *= scalar;
25    }
26    public void ToUnit()
27    {
28        double magnitude = Length;
29        X /= magnitude;
30        Y /= magnitude;
31    }
32    public static Vector2D FromRoad(Road road)
33    {
34        Vector2D vector = new Vector2D();
35        vector.X = road.To.Position.X - road.From.Position.X;
36        vector.Y = road.To.Position.Y - road.From.Position.Y;
37        return vector;
38    }
39 }
40 }
```

Figure 10.12: MathExtension Klasse

10.7 Viewport

Viewport klassen har til formål at sætte rammerne for området hvori man kan tegne elementerne i vejnettet.


```
1 public readonly int GridLength = 1000;
2 public readonly int GridSize = 16;
3 public readonly int EntitySize = 12;
4 public readonly int NodeSize = 8;
5
6 public Project Project;
7 public Point HoverConnection = new Point(-1, -1);
8 public Point MousePos = new Point(0, 0);
9 public Point GridPos { get { return GetGridPos(); } }
```

Figure 10.13

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 projekt og en række parametre set i figur 10.13. `ReadOnly` variablerne for `GridLength`, `GridSize`, `EntitySize` og `NodeSize` er størrelserne for de visuelle objekter på Viewporten.

Der bliver instansieret en variabel af typen `Project` da Viewporten skal kunne repræsentere projektet. `HoverConnection` visualiserer den forbindelse man prøver at lave mellem to objekter. `MousePos` indikerer det aktuelle punkt hvor musen befinder sig i gitteret, 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 10.14

Som sagt kan `Viewport` indikere koordinaterne til indsatte objekter og dette gør den igennem metoden `GetObjByGridPos` som set i figur 10.14. Metoden tjekker for alle `Node`, `LightController` og `Destination` om deres position er lig `GridPos`. Hvis den har fundet en, så returnere den det fundne objekt.

10.8 Pathfinder

For at forklare implementationen af Astjerne, 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 resterende 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.LongCount() > 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("There isn't any route");
26 }
```

Figure 10.15: FindPath metoden

Den første metode er **FindPath**, der ses på figur 10.15, 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             neighbor.CalculateEstimate(current, edge, End, MaxSpeed);
10            Open.Add(neighbor);
11            if (neighbor.Cost <= current.Cost + edge.Cost)
12                neighbor.Previous = current;
13        }
14    }
15 }
```

Figure 10.16: EstimateNeighbors metoden

I EstimateNeighbors metoden, der kan ses på figur 10.16, kigger vi på naboerne, til den Vertex der bliver givet gennem parameteret. Hvis naboen allerede er evalueret, skal vi ikke gøre det igen, ellers bliver de evalueret gennem metoden CalculateCostEstimate, der ligger i selve Vertex klassen. Sidst tjekkes der om naboens Cost er bedre end den nuværende, hvor vi så vil sætte dens Previous reference til at være den nuværende Vertex, så vi senere kan finde vejen tilbage igen.

```
1 public void CalculateCostEstimate(Vertex previous, Edge edge,
2                                 Vertex end, int maxSpeed)
3 {
4     Cost = previous.Cost + edge.Cost;
5     double heuristic = MathExtension.Distance(this.Position,
6                                              end.Position) / maxSpeed;
7     Estimate = Cost + heuristic;
8 }
```

Figure 10.17: CalculateCostEstimate metoden

CalculateCostEstimate metoden, der er vist på figur 10.17, udregner hvad kosten er fra start til dette punkt, og estimerer med heuristiken hvad den mindst mulige kost kunne være fra her til slut punktet.

```
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 10.18: TracePath metoden

Sidst har vi på figur 10.18 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.

10.9 Funktionelle

10.9.1 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 10.19: ToolController metoden

På figur 10.19 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 == MouseButton.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 10.20: ViewportClick metoden

Derudover bliver click eventen på Input lageret af Viewporten i constructoren sat til at blive håndteret af metoden ViewportClick. ViewportClick der kan ses på figur 10.20 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 10.21: ToggleTool metoden

For at sikre at der ikke er “valgt” flere værktøjer på samme tid, kaldes metoden på figur 10.21 ToggleTool, 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 variabelen. 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 10.22: Add metoden

Add metoden der ses på figur 10.22 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 `NodeType`en 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 10.23: SetNodeType metoden

`SetNodeType()` som er vist på 10.23, 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 `NodeType`en 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 10.24: AddRoad metoden

`AddRoad()` som kan ses på figur 10.24, bruges til at tilføje en vej mellem 2 noder, derfor checkes der først om object på den nuværende position 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 nuværende position 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 nuværende position i gitteret.

Hvis brugeren holder "Shift" nede imens, vil programmet sætte `_firstNode` til den nuværende `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.

10.9.2 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 **FileHandler**en. 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 10.25: OpenProject metoden

Metoden **OpenProject**, der vises på figur 10.25, kan åbne et eksisterende projekt, når brugeren trykker på Open Project i 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 tjekker om der om brugeren har valgt 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.


```

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

```

Figure 10.26: SaveProject metoden

På figur 10.26 ses metoden til at gemme et projekt. Igen benytter programmet af en try-catch-finally, hvor programmet i try fasen laver en ny instans af `SaveFileDialog`, på den måde kan programmet gemme et projekt som man har arbejdet på. Programmet er sat op at den skal gemme projektet som tsp fil (traffic simulation project). Hvis det ikke lykkes, så vil den kaste en exception med fejlen der er sket. Finally fasen vil den så frigøre ressourcerne igen.

MANGLER: Noget om åbning og gemning af `SimulationData`

10.9.3 Simulation

10.10 Vehicle

`Vehicle` klassen repræsenterer 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 10.27: Vehicle Constructor parametre

Constructoren til `Vehicle` tager imod en række forskellige parametre som ses på figur 10.27, og udfører nogle opgaver, så køretøjet er klar til at køre. Som figur 10.27 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** variabelen **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 10.28: Drive metoden

Som afsnit 10.9.3 beskriver, bliver **Drive** kaldt for hver **Vehicle** instans, hver gang simulationen tager et step. I **Drive** metoden som vist på figur 10.28, 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 konstanten **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        && !(incomingVehiclesCount == 1
14        && _currentRoad.To.IncomingVehicles.Contains(this)))
15        return 0;
16    else if (vehicleInfront != null)
17        return vehicleInfront.Speed;
18    else
19    {
20        if (Type.MaxSpeed > _currentRoad.Type.Speed)
21            return _currentRoad.Type.Speed;
22        else
23            return Type.MaxSpeed;
24    }
25 }

```

Figure 10.29: GetSpeed metoden

På figur 10.29 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 variabelen `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, bliver hastigheden sat lig med køretøjet. 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 10.30: Move metoden

Figur 10.30 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 10.31.

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

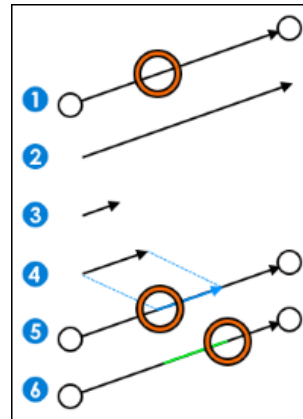


Figure 10.31: TranslateVehicle

```

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 10.32: 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 noden 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 deaktivere, ellers hvis slutnoden 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 noderne 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** (vigepligt).

11

Diskussion

12

Konklusion

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 simuleringen. Dette vil have en positiv effekt på programmet, da programmet ikke vil kræve for mange ressourcer for at lave en simulering.

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 håndteret 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.

En anden måde at kigge på det kunne også være at man kunne optimere processen bag simulering, ved at sætte GPU'en til at beregne simulering beregningerne. Vores gruppe har benyttet CPU'en til at lave beregningerne, men vi kunne have benyttet GPU'en for at gøre processen hurtigere.

Når vores program kører i simulationen, så bilerne som der er i simulationen kender ikke noget til de modkørende. Det er grunden til at bilerne ikke har muligheden for at overhale hinanden. Den måde vi har sat vejene op, er ved at tegne to veje lige ovenpå hinanden, bare i modsatte retninger. vejene er heller ikke sammensat, så der er plads til at kunne overhale. Så hvis der er en bil kørende bag en der kører langsommere, så vil han sætte farten ned og blive bag ved bilen.

Bibliography

- [1] Pshko Aziz. Aalborg kommune teknisk forvaltning, 2016.
- [2] Christopher A. Chung. *Simulation Handbook - A Prictical Approch*. CRC Press LLC, 2004.
- [3] Altair Engineering. About visual solutions, incorporated. <http://www.vissim.com/company.html>, 2016.
- [4] Carsten Broder Hansen. Dtu - forskning. <http://www.transport.dtu.dk/Forskning>, 2015.
- [5] 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.
- [6] 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.
- [7] 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.
- [8] Matthew Barth og Kanok Boriboonsomsin. Traffic congestion and greenhouse gases. <https://escholarship.org/uc/item/3vz7t3db>, 2009.
- [9] Michael Knørr Skov og Karsten Sten Pedersen. Trafikprop. flere veje vil skabe større vækst. http://www.cowi.dk/menu/tema/infrastruktur-2030/cowi-i-mediernes/Documents/Veje%20skaber%20v%C3%A6kst_Politiken%20analyse%2024052014.pdf, 2014.
- [10] Henrik Papsø. Trafikpropper kan koste menneskeliv. <http://www.tveast.dk/artikler/trafikpropper-kan-koste-menneskeliv>, 2007.

-
- [11] Introduction to A*. <http://www.redblobgames.com/pathfinding/a-star/introduction.html>, 2016. Accessed: 06/03/2016.
 - [12] Kenneth H. Rosen. *Discrete Mathematics and its Applications*. McGraw Hill, 7. global edition, 20113.
 - [13] Kenneth H. Rosen. *Discrete Mathematics and its applications*. Kamala Krithivasan, 1. edition, 2015.
 - [14] Paul Ross. Traffic dynamics. Technical Report 6, Traffic Systems Division, Federal Highway Administration, August 1988.
 - [15] 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.
 - [16] Vejdirektoratet. Længden af offentlige veje. http://www.vejdirektoratet.dk/DA/viden_og_data/statistik/vejeneital/1%C3%A6ngdeoffentligeveje/Sider/default.aspx, 2016.
 - [17] Vissim.com. http://www.vissim.com/downloads/doc/VisSim_UGv80.pdf, 2015. Accessed: 17-03-2016.

A

Appendix