AFSLUTNINGSPROJEKT

COPENHAGEN BUSINESS ACADEMY

AKADEMIUDDANNELSEN I IT

Jens Ørnfeldt Gaul

JUNI 2021

VEJLEDER: Kasper Videbæk

Indholdsfortegnelse

1 Indledning 4
1.1 Problemområde 4
1.2 Problemformulering 4
1.3 Afgrænsning af opgaven 4
2 Design og Krav 5
2.1 Beskrivelse af et transportsystem 5
2.2 Datamodel 5
2.2.1 Berlin 5
2.3 Valg af søgealgoritme 9
2.4 Søgningen efter en optimal heuristik. 9
2.5 Selve applikationen 11
2.5.1 Søgning på stationer 11
2.5.2 Præsentation af resultat 12
2.5.3 Design 12
3 Valg af teknologier 13
3.1 Database 13
3.2 Programmeringssprog og platform 13
3.2.1 Valg af web teknologi 14
4 Implementering 15
4.1 Databasen 15
4.1.1 NetworkNodes 15
4.1.2 Edges og Timetables 15
4.1.3 Implementering af datamodellen i C# 15
4.2 Beregning af afstand og tider 16
4.2.1 Gåafstande 16
4.2.2 Hjælpeklasse til tidsberegning 16
4.3 A* algoritmen 17
4.3.1 Input til A* 17
4.3.2 Output fra A* 18
4.3.3 Datastrukturer 18
4.3.4 Beskrivelse af algoritmen 18
4.3.5 A* baglæns 19
4.3.6 Beregning af heuristik (optimal hastighed) 20
4.3.7 Store ventetider i transportsystemet 21
4.3.8 Indeks i databasen 22

4.4 Viderebehandling af data efter A* 23
4.4.1 Viewmodel til strækning (RoutePath) 23
4.4.2 Viewmodel til afgangstavler (DepartureBoard) 24
4.5 Web front-end implementering 24
4.5.1 Twitter typeahead 24
4.5.2 Kort 24
4.5.3 API 24
5 Test af A stjerne 25
6 Resultater af søgninger 26
7 Konklusion 27
8 Perspektivering 27
9 Links til repository 28
10 Litteraturliste 28
11 Bilag 28

1 Indledning

Alle kender situationen med at skulle finde vej med tog, bus eller undergrundsbane. Hvordan kommer jeg nemmest fra a til b, hvor skal jeg skifte bus, hvilken retning skal jeg køre med toget og hvornår er jeg fremme?

1.1 Problemområde

Denne opgave beskæftiger sig med at finde korteste vej gennem et *transportsystem*.

Et *transportsystem* er en samling af *ruter* der køres mellem forskellige stationer med f.eks. bus, S-tog, undergrundstog eller let bane. En *rute* er beskrevet ved at have en start station, en række stop undervejs og en endestation, og for hvert stop, er der en ankomst- og afgangstid. Ruten har en rutebetegnelse, et navn på endestationen og kører efter en bestemt køreplan.

Offentlige transportselskaber udgiver deres køreplaner og andre data i et format kendt som GTFS[1]. Det er bygget over en kompleks datamodel, da det skal kunne indeholde data fra mange forskelligartede systemer. Jeg har derfor fundet et datasæt som er udgivet til forskningsbrug. Dette datasæt er betydeligt mere enkelt, og beskriver en by med tilhørende stationer, strækningsplaner, køretider og dækker en køreplan for en uge (Bilag 1).

Denne opgave laves over Berlin, Tyskland. Grunden er, at jeg kender byen godt, stationsnavne, lokationer giver derfor mening for mig, og en rejse fra a til b, kan jeg relatere til.

1.2 Problemformulering

Ud fra disse forskningsdata skal det undersøges om det er muligt at konstruere en datamodel, der gør det muligt at lave effektive søgninger.

I opgaven skal det også undersøges, hvilken algoritme der skal bruges, så søgninger fremstår hurtige og korrekte for brugerne.

Kan det derefter lade sig gøre at fremstille en webapplikation, der kan anvendes på forskellige enheder, hvor en bruger har mulighed for at søge en rejse i Berlin. Brugeren skal kunne søge blandt stationer/stoppesteder i byen og så angive en afgangs- eller ankomsttid på en uge dag. Applikationen skal derefter finde den hurtigste vej og præsentere en rejseplan for brugeren.

1.3 Afgrænsning af opgaven

Der arbejdes ikke med realtime data i opgaven, forstået på den måde at køreplaner og rutedata er statiske.

[1] GTFS = General Transit Feed Specification: https://gtfs.org/

2 Design og Krav

I dette afsnit beskrives de overordnede krav, som en uddybning af problemformuleringen. De spørgsmål som findes i problemformuleringen uddybes yderligere og der kommes med mere konkrete krav til slutproduktet.

2.1 Beskrivelse af et transportsystem

En del offentlige transportselskaber vælger at offentliggøre deres køreplan i et format kendt som GTFS (General Transport Feed Specification). Formatet understøttes bl.a. af Google, og bruges til at vise data i Google Maps. Af GTFS Transit APIs/Reference (https://gtfs.org/reference/static) findes oversigten over den datamodel der benyttes, indeholdende 17 tabeller, og over 130 forskellige felter, med tilhørende forklaring og regler.

Denne datamodel er ret kompleks, da den er udviklet til at kunne indeholde alverdens transportsystemer, og samtidig kan der være tilfælde hvor et transportsystem er et helt land, eller det kan være, af at hver transport leverandør i en by, udgiver hvert deres GTFS-feed.

For lettere at kunne sammenligne forskellige transportsystemer, har en række forskere gereret et 'forsimplet datasæt' og det dækker en typisk uges 'almindelig drift'. Den fulde beskrivelse findes som bilag 1. Jeg vil her ikke gå nærmere ind på de metoder der er brugt på at transformere GTFS-feed'et til de data som jeg bruger som udgangspunkt for min opgave, det kan læses i detaljer i bilag 1, og er egentlig irrelevante for opgaven her.

2.2 Datamodel

Et transportsystem kan bedst beskrives med en graf.

En graf består af noder (*verticies*) og er forbundet med kanter (*edges*). En kant kan have en pris/vægt og det når det er tilfældet, kaldes det en vægtet graf.

Noderne i et transportsystem er stoppesteder/stationer og kanterne er de strækninger som forbinder de forskellige noder. Om det er skinner, asfalt eller vand er underordnet. Prisen/vægten kan være udtryk for rejsetid/afstand eller billetpris.

Grafen i et transportsystem er en ikke komplet graf, da alle noder ikke er forbundet med alle kanter. Grafen er bi-direktional, da er kan være forskellige rejsetider mellem noder frem og tilbage. Grafen vil desuden være dynamisk, idet den ændrer sig med tiden grundet køreplanen. Om aftenen vil der f.eks. være kanter i grafen der ikke betjenes, eller der kan være kanter der kun køres med mindre intervaller.

I mit transportsystem er det udelukkende tider i en køreplan der kigges på, derfor vil vægten i min graf være tid.

2.2.1 Berlin

Datasættet for Berlin består af 4601 stoppesteder med 12.079 forbindelser, dækkende en 30 km radius fra centrum, ca. 2.800 km² (bilag 1, tabel 3 og 4). Det er beskrevet i 14 tabeller, som er en kombination af komma separerede lister, sqlite databaser, json og zip filer.

Følgende tabeller fra datasættet er benyttet i denne opgave:

Titel	Beskrivelse
network_nodes.csv	Beskrivelse af de enkelte stop, bl.a. geo koordinater og ID-nummer
network_walk.csv	Gå forbindelser mellem stationer
network_temporal_week.csv	Den fulde køreplan, beskrevet som ture mellem stops på tid etc.
week.sqlite	Sqlite database indeholdende køreplan for én uge.

Den centrale tabel for min datamodel er *network_temporal_week.csv*, og danner udgangspunkt for konstruktionen af grafen. Det gør den, da den indeholder data om den graf jeg skal konstruere, samt tider i køreplanen.

Tabellen er struktureret således:

Felt navn	Beskrivelse
From_stop_I	Id-nummer på afgangs stop
To_stop_I	Id-nummer på ankomst stop
Dep_time	Afgangstiden fra From_stop_I
Arr_time	Ankomsttiden til To_stop_I
Route_type	Transport middel Id (se bilag 1, tabel 2)
Trip_I	Id-nummer på turen
Seq	Rækkefølgen af stop på turen
Route_I	Id-nummeret på en rute (fx busrute)

Figur 1: network_temporal_week.csv

```
from_stop_I;to_stop_I;dep_time_ut;arr_time_ut;route_type;trip_I;seq;route_I
6507;6502;1461552300;1461552660;3;1;1;1
6502;5491;1461552660;1461553140;3;1;2;1
5491;5534;1461553140;1461553560;3;1;3;1
5534;5572;1461553560;1461553860;3;1;4;1
5572;5477;1461553860;1461553980;3;1;5;1
5477;7199;1461553980;1461555120;3;1;6;1
6507;6502;1461638700;1461639060;3;1;1;1
6502;5491;1461639060;1461639540;3;1;2;1
5491;5534;1461639540;1461639960;3;1;3;1
5534;5572;1461639960;1461640260;3;1;4;1
5572;5477;1461640260;1461640380;3;1;5;1
5477;7199;1461640380;1461641520;3;1;6;1
6507;6502;1461725100;1461725460;3;1;1;1
6502;5491;1461725460;1461725940;3;1;2;1
5491;5534;1461725940;1461726360;3;1;3;1
```

Tabellen består af flere mange-til-mange relationer. Tuplen (from_stop_I, to_stop_I) optræder mange gange, og det samme gør tuplen (Trip_I, Route_type, Route_I). For at bryde disse mange-til-mange relationer, splittes tabellen op i to tabeller for at beskrive grafen med vægte og afgangstider.

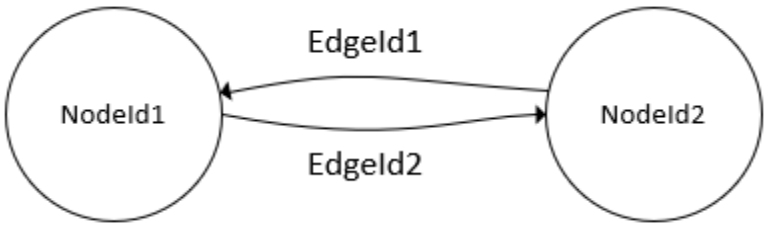
Der findes grundlæggende to metoder at beskrive en graf. *Adjacency list* eller *Adjacency matrix*. En adjacency list beskriver naboer til en node, som en simpel liste struktur, hvorimod en adjacency matrix laves som en 2 dimensional matrix. Matrixen her hurtig at slå op i, men fylder N^2, mens listen fylder væsentlig mindre, idet kun naboer der er forbundet, er beskrevet i listen for hver node. I denne model benyttes adjacency list, da grafen er meget tynd[2].

Herunder følger en diskussion af de primære felter i datamodellen, se Figur 4: Datamodel.

Filen *network_nodes.csv* er en beskrivelse af alle stop i byen. Hver stop har et unik ID i tabellen, og tabellen indsættes næsten direkte i min tabel *NetworkNodes*. Der findes derudover geografiske informationer, som præcist stedbestemmer stoppet i byen.

Tabellen *Edges*, er den *adjency list* som beskriver den vægtede graf. Hver node har et Id-nummer som kommer fra NetworkNodes, og hver kant i den bi-direktionale graf har et unikt EdgeId, og dermed kan bruges som primær nøgle. Det bemærkes at der *kan* være en kant, både frem og tilbage.

[2] At en graf er tynd, betyder at der er meget få kanter til hver node.



Figur 2: Simpel graf

Ud fra *Edges* tabellen, er det nu muligt at lave en søgning, som kan finde alle naboer til en bestemt node, og herefter få returneret det EdgeId, som svarer til kanten.

TimeTables tabellen konstrueres ud fra *network_temporal_week*, som indeholder alle afgange fra alle stationer i den uge, som datasættet dækker. Ud fra de to stations id'er indsættes i stedet EdgeId.

TimeTables tabellen, gør det nu muligt ud fra et EdgeId, at få (alle) afgangstider på en bestemt kant i grafen.

For at skaffe informationer om bus/tog ruter skal tabellen *trips* i week.sqlite filen undersøges. Tabellen indeholder også en del redundant data. Fx findes teksten til 'headsign' mange gange.

trip_I	trip_id	route_I	service_I	direction_id	shape_id	headsign
Filter	Filter	Filter	Filter	Filter	Filter	Filter
273	273	585	1997	NULL	NULL	Wegendorf, Siedlung
274	274	573	1991	NULL	NULL	Wegendorf, Siedlung
275	275	596	2281	NULL	NULL	Wegendorf, Siedlung
276	276	596	2282	NULL	NULL	Wegendorf, Siedlung
277	277	596	1989	NULL	NULL	Strausberg, Lustgarten
278	278	596	1989	NULL	NULL	Strausberg, Lustgarten

Figur 3: Uddrag af week.sqlite, tabellen trips

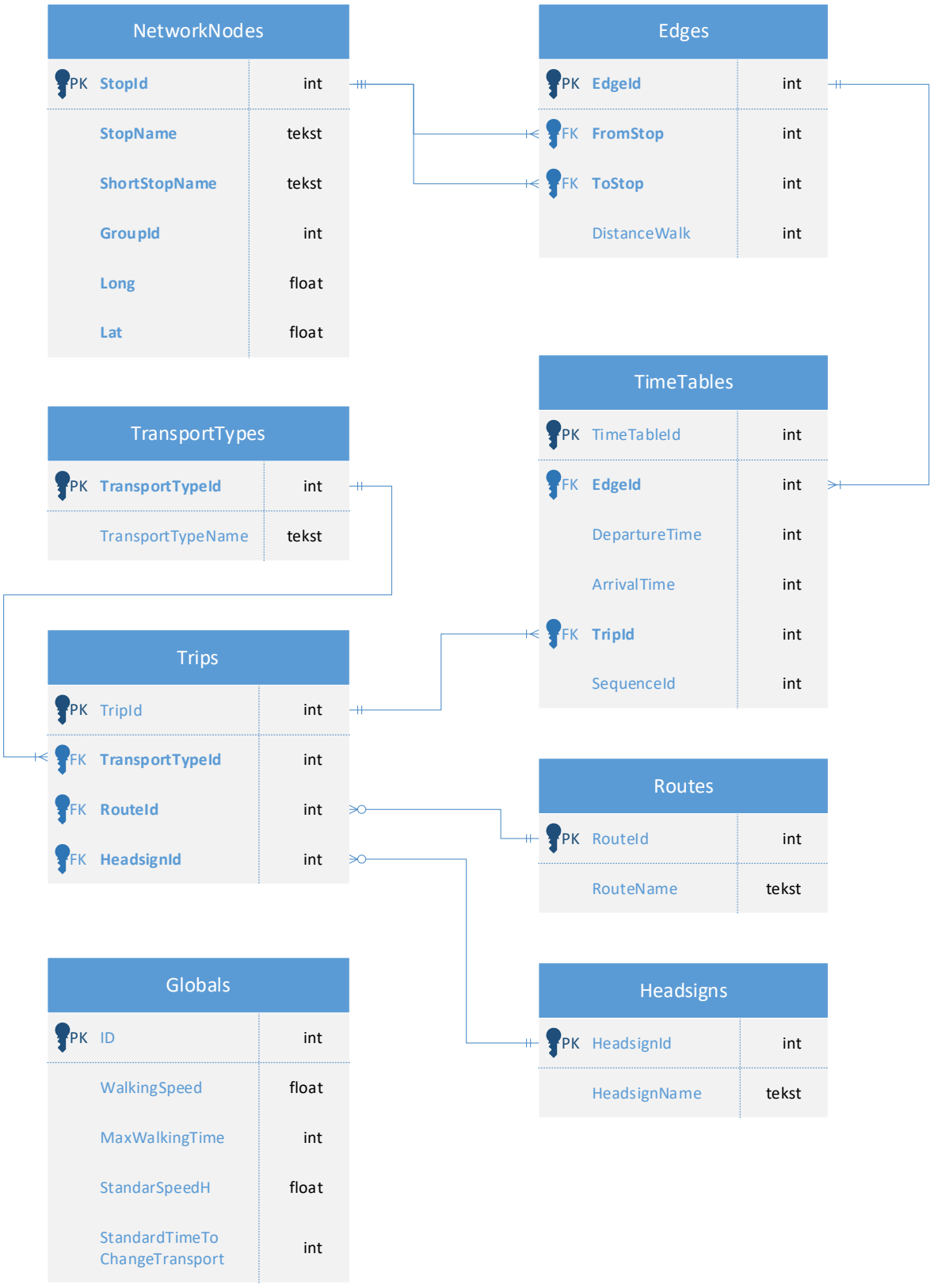
Trip_I beskriver en rute som køres mellem to ende stop. Ruten har en rutebetegnelse (route_I) fx bus nummer 3, og et skilt med endestationen (headsign). En rute kan godt køres flere gange, men en rute kører altid til en bestemt endestation, og har en bestemt rutebetegnelse.

Hvis man kigger tilbage på Figur 1: network_temporal_week.csv, så vil man se at fx både Route_type, Route_I, ligeledes findes der.

Ud fra de data som findes i week.sqlite tabellen, konstrueres en *Trips* tabel, som (med undertabeller) indeholder data for ruter i netværket. Således findes en liste over alle skilte med endestationer i tabellen *Headsigns*, og rutebetegnelser i tabellen *Routes* samt transporttyper i tabellen *TransportTypes.* Hermed sikres at data ikke er redundante.

Ud fra denne datamodel er det nu muligt at lave søgninger til at finde korteste vej, og data redundans er fjernet.

Desuden er der lavet en tabel, globals. Denne tabel bruges til globale variable i programmet, som jeg kan få behov for at ændre i løbet af udviklingen.



Figur 4: Datamodel

2.3 Valg af søgealgoritme

Ud fra den valgte datamodel, som er en bi-direktional, vægtet, ikke komplet graf, skal findes en algoritme som kan finde den hurtigste vej gennem grafen.

Er hurtigste = korteste? Ikke nødvendigvis, men jeg kommer til at bruge disse begreber i flæng i dette afsnit. Når jeg når til implementeringen, vil det stå klart, at al regning foregår i sekunder, da algoritmerne er låst op på en køreplan. I dette afsnit er det det samme.

Der findes grundlæggende to søgeteknikker når det handler om grafer, nemlig dybde først søgning (depth first search, DFS), eller bredde først søgning (breadth first search, BFS). Begge algoritmer starter i en udvalgt node og løber gennem grafen, indtil slut noden er fundet, eller ingen vej er fundet.

DFS, fungerer ved at besøge én gren af grafen helt til bunds, førend den søger på den næste gren, som ikke allerede er besøgt. I et scenarie hvor vi skal finde kortest vej, er det ikke nødvendigvis en god strategi, idet algoritmen risikerer at undersøge alle andre nabogrene helt, for så at finde at slutnoden var en nabo til startnoden, i den sidste gren der besøges.

BFS, fungerer ved at undersøge alle kanter til én node, hvorpå vi rykker ud til næste node, og undersøger alle kanter, der ikke allerede er undersøgt. BFS egner sig godt til korteste vej, og algoritmen er forfinet med en del tilpasninger i nye og afledte algoritmer.

En af de mest kendte er Dijkstra's algoritme. Den besøger samtlige noder i grafen, indtil den finder den korteste vej. Den bruger en prioritets kø til at udvælge noder der har den korteste vej, indtil videre, for på den måde at udvælge den korteste vej. Problemet er her, at Dijkstra ikke ved hvilken vej der er den rigtige, den prioriterer udelukkende på den korteste vej, og det er ikke nødvendigvis den rigtige vej. Det skal dog siges, at Dijkstra altid finder den hurtigste vej, men det tager længere tid for algoritmen.

Der findes også en metode som kaldes bi-direktional Dijkstra. Den løber på den måde, at der startes en Dijkstra søgning fra hver ende af, og så er korteste vej, der hvor de to søgninger mødes. Det fungerer fint i en statisk graf, men slet ikke i en graf der er dynamisk, for hvilken ankomsttid skulle man sætte som udgangspunkt for den baglæns søgning? Det kan derfor ikke bruges i denne opgave.

En algoritme der kan være bedre egnet, er A stjerne. Den bygger også på BFS princippet, og er en videre udvikling af Dijkstra's, på den måde at den ved hjælp af en *heuristik* forsøger at gætte den korteste vej. På den måde vil algoritmen bedre kunne navigere i en graf, og finde en hurtigere finde vej, hvis en sådan findes. Heuristikken hjælper med andre ord, algoritmen med at prioritere den rigtige retning. Algoritmen ved jo ikke som udgangspunkt hvad der er rigtig eller forkert retning, men ved at beregne den forventede tid/afstand det tager at komme frem, vil algoritmen kunne prioritere den vej, som er hurtigst, vha. en prioritets kø.

Forudsætningen for at A* er hurtigere end Dijkstra er at der findes en optimal heuristik.

2.4 Søgningen efter en optimal heuristik.

Hele søgningen er bygget op omkring tider, og derfor bliver heuristikken en funktion der udtrykker tid.

Heuristikken beskrives som funktionen $f(v) = cost\ (v) + h\ (v, d)$. v er den node vi er kommet til, og d er destinationen. $cost\ (v)$ er den udregnede tid der er brugt til at komme fra start til v.

Eftersom at vægtene i grafen betegner tider, skal $h\ (v, d)$ være den tid det tager at tilbagelægge afstanden mellem v og d.

Der findes mange eksempler på hvordan en optimal heuristik findes, men eksemplerne er alle bygget op omkring *statiske* grafer og mange beskæftiger sig med, at finde den korteste vej for en bil. I de tilfælde er

der ikke nogen køreplan at tage hensyn til, men udelukkende max hastighed for området. Elles refereres der en del til korteste vej i spil.

Jeg har søgt en del omkring emnet på internettet, og de fleste er enige om at følgende udsagn er retvisende for at finde korrekt heuristik i en statisk graf.

Her er hvad Amit Patel, Stanford University skriver om netop heuristik i A*:

- At one extreme, if `h(n)` is 0, then only `g(n)` plays a role, and A* turns into Dijkstra's Algorithm, which is guaranteed to find a shortest path.
- If `h(n)` is always lower than (or equal to) the cost of moving from `n` to the goal, then A* is guaranteed to find a shortest path. The lower `h(n)` is, the more node A* expands, making it slower.
- If `h(n)` is exactly equal to the cost of moving from `n` to the goal, then A* will only follow the best path and never expand anything else, making it very fast. Although you can't make this happen in all cases, you can make it exact in some special cases. It's nice to know that given perfect information, A* will behave perfectly.
- If `h(n)` is sometimes greater than the cost of moving from `n` to the goal, then A* is not guaranteed to find a shortest path, but it can run faster.
- At the other extreme, if `h(n)` is very high relative to `g(n)`, then only `h(n)` plays a role, and A* turns into Greedy Best-First-Search.

Figur 5: Amit Patel, Standford

Ved at søge på internettet om valg af heuristikker til A* i transportsystemer, bliver det hurtigt klart at det ikke er en triviel opgave at vælge en korrekt heuristik.

De fleste er dog enige om at fremgangsmåden som Amit Patel beskriver (ikke at de referer til ham) er korrekt.

Hvis vi forudsætter at der ingen ventetid er i vores transportsystem, så toget går når vi står på perronen, bussen kører med det samme, så vil Amit Patels antagelser om at finde en brugbar heuristik være gangbar. Men det holder ikke når der introduceres ventetider.

Eksempel: Den hurtigste vej er at tage et ekspres tog som går én gang i timen. Vi er bare ankommet 55 minutter før næste afgang. Derfor vil A* begynde at afsøge alle mulige andre veje, for til sidst at finde vejen med ekspres toget. Dermed bliver algoritmen meget langsom. (Merrifield, 2010, pp. 35-37)

En anden udfordring er, at de forskellige transportformer har forskellige hastigheder. Så der vil muligvis kunne findes en korrekt heuristik for S-tog, eller bus alene, men hvad sker der, når søgningen kombinerer de to?

Ved at undersøge de forskellige transportformer der anvendes i Berlin, er jeg kommet frem til følgende hastigheder.

Transportform	Gennemsnitshastighed	Max hastighed (km/h)
S-tog	38	100
U-bahn	24	72
Bus/Tram	20	80
Gå	5	5

Tabel 1: Hastigheder i Berlin transportsystem

Hvordan den optimale hastighed til heuristiken findes, beskrives senere i opgaven.

2.5 Selve applikationen

Den applikation som brugen skal møde, er en webside i en browser.

Herfra skal det være muligt for en bruger at kunne finde vej mellem to stationer. Følgende oplysninger skal indtastes:

- Afgangsstation
- Ankomststation
- Dag og tidspunkt (eller afgang nu)
- Om det er afgangs- eller ankomsttid

Der skal som sådan ikke direkte indtastes et stationsnavn, men ved at brugeren begynder en indtastning, skal der foreslås navne. Og det leder direkte videre til det næste afsnit, vedr. søgningerne.

2.5.1 Søgning på stationer

Det viser sig ved nærmere eftersyn, at nogle stationer er så store, at de har flere id-numre. En station som Alexander Platz, dækker adskillige U-bahn linjer, S-tog, busser og Trams. Hvert af disse stop har et unikt nummer, og hver stop har sit eget navn.

Når en bruger søger på Alexander Platz, så ved brugeren jo ikke om ruten skal starte med tog eller bus, og kommer derfor måske til at vælge det forkerte udgangspunkt.

Nedenstående Tabel 2: Søgning på Alexander, viser tydeligt problemet med at flere stoppesteder har næsten samme navn.

```
Line 1769: 1880;52.521516;13.411272;S+U Alexanderplatz Bhf (Berlin)
Line 1771: 1882;52.522394;13.414498;U Alexanderplatz (Berlin) [Tram]
Line 1788: 1900;52.521831;13.411238;S+U Alexanderplatz Bhf/Dircksenstr. (Berlin)
Line 1790: 1902;52.521067;13.411258;S+U Alexanderplatz Bhf/Gontardstr. (Berlin)
Line 1793: 1905;52.523101;13.410969;S+U Alexanderplatz Bhf/Memhardstr. (Berlin)
Line 1821: 1933;52.517749;13.418135;Alexanderstr. (Berlin)
Line 1842: 1954;52.522081;13.413604;S+U Alexanderplatz (Berlin) [U2]
Line 1843: 1955;52.521611;13.413117;S+U Alexanderplatz (Berlin) [U5]
Line 1844: 1956;52.521622;13.41213;S+U Alexanderplatz (Berlin) [U8]
Line 1845: 1957;52.522886;13.41469;U Alexanderplatz [Bus]
Line 1846: 1958;52.521801;13.408423;S+U Alexanderplatz (Bln) [Bus K.-Liebknecht-Str]
Line 1849: 1961;52.520621;13.414676;S+U Alexanderplatz/Grunerstr.(Bln) [Grunerstr.]
Line 1850: 1962;52.520321;13.415713;S+U Alexanderplatz/Grunerstr.(Bln) [Alexanderstr.]
Line 1850: 1962;52.520321;13.415713;S+U Alexanderplatz/Grunerstr.(Bln) [Alexanderstr.]
```

Tabel 2: Søgning på Alexander

For at imødekomme dette, skal der findes en metode der kan hjælpe brugeren, så der kun findes et Alaxander Platz stop i en søgning.

Desuden findes der stop, som hedder næsten det samme, men måske blot er i hver sin ende af en mindre plads, eller på hvert sit gadehjørne. For at forsøge at give brugeren den bedste oplevelse, skal der findes en løsning på dette problem.

Da opgaven dækker Berlin, er det ikke oplagt at der findes en funktion for at anvende brugerens aktuelle GPS-position, så det er udeladt.

2.5.2 Præsentation af resultat

Når søgningen er færdig, skal brugeren have præsenteret en rute med beskrivelse, som gør det muligt at finde vej.

Der skal være oplysninger nok til at brugeren får udførlig vejledning om ruten:

- Start stoppested
- Afgangstid
- For hvert ben på ruten skal der være oplyst følgende:
    - Afgangstid
    - Rute betegnelse
    - Transport form
    - Navnet på endestationen for ruten (*retnings skilt*)
    - Ankomsttid
    - Evt. liste over de stop som benet kører gennem
    - Skifte tid eller gå tid/afstand
- Ankomsttid for slutdestination
- Ankomststationen
- Samlet transporttid
- Et kort der viser stoppestederne

2.5.3 Design

Applikationen skal tilpasse sig 3 forskellige formater:

- Telefon, højkant
- Telefon, vandret
- Computerskærm, vandret

3 Valg af teknologier

Herunder beskrives hvilke teknologier der vælges på baggrund af de beslutninger der er taget under analysen.

3.1 Database

Til at repræsentere de data der er nødvendige i modellen, kigger jeg lidt på mængden af data som findes i køreplanen for én uge.

- 4.601 stationer/stoppesteder
- 47.930 forbindelser mellem stationerne
- Ca. 6.7 millioner afgange fra et stoppested i løbet af perioden
- 620 forskellige ruter, med 1241 forskellige endestationer

I sig selv ikke nogen kæmpe datamængde, som vil kunne håndteres af alle kendte DBMS[3], der er dog et par oplagte, idet de er 'gratis':

1. MySQL
2. Sqlite
3. Microsoft SQL Express

Ad 1:
MySQL er kendt stof fra undervisningen, og kræver at der kører en server i baggrunden. Den har en høj grad af sikkerhed, og virker godt ved både skrive og læsninger.

Ad 2:
Sqlite er serverless, dvs. der skal ikke kører en server i baggrunden. Sikkerheden er ikke særlig god, og skrivninger i databasen låser *hele* skemaet, mens det foregår. Men i mit tilfælde, skal der ikke skrives data under brug, det klares i en helt separat applikation, der danner databasen inden den skal bruges.

Ad 3:
Microsoft SQL Express, er som de to andre et gratis produkt, og en database lavet i SQL Express, lader sig næsten uden videre portere til den store server udgave af Microsoft SQL. Sikkerheden er god, og skalerbarheden ligeledes.

Mit valg falder på Sqlite, da den er serverless, og de databaser som mit datasæt udspringer fra, ligeledes er fra Sqlite. Da der ikke skal opbevares nogle brugerdata, er sikkerhed/kryptering ikke noget problem.

3.2 Programmeringssprog og platform

Der er mange muligheder for platform til webapplikationen, og bagvedliggende programmeringssprog. Hvis jeg besøger de sprog, jeg mestrer, kommer jeg frem til følgende:

1. JavaScript
2. Python
3. C#

Ad 1, JavaScript:
Jeg kunne vælge at skive en komplet webapplikation i fx node.js, men mit kendskab er begrænset til netop node.js. Desuden syntes jeg at JavaScript mangler den typestærke programmering, som ofte gør at det bliver vanskeligt at holde sproget rent og klart.

[3] DataBase Management System

Ad 2, Python:
Det er et hurtigt sprog at skrive prototyper i. Jeg vælger at lave den del af min applikation der skaber databasen i Python. Det er også muligt at lave en helt fuld funktionsdygtig web-server med fx flask, men min erfaring er begrænset og jeg savner ligeledes den stærke klasse nedarvning og typer.

Ad 3, C#:
Microsoft stiller et fuldt framework til rådighed, med front-end og back-end. Der er flere under teknologier at vælge imellem. Den nyeste er ASP.CORE med Blazor, hvor det er muligt at skrive C# kode direkte i klienten. Der findes derudover ASP.CORE med Razor eller ASP.Net framework MVC. Jeg har godt kendskab til sidstnævnte, og vælger dette, selvom det er en smule ældre teknologi. Jeg vælger den sikre og kendte vej. Derudover er C# et typestærkt sprog, og understøtter klassehierarkiet, med nedarvning og ind-kapsulering.

3.2.1 Valg af web teknologi

For at muliggøre skalering til de 3 skærm størrelser jeg har valgt, vil jeg bruge Bootstrap. Bootstrap har et system, der gør det muligt nemt at skrive html og css kode, som hjælper med at skalere sider. Alternativet til Bootstrap, vil være Vanilla JavaScript med css media tags, men det gør det vanskeligere at håndtere skalerbarhed.

4 Implementering

4.1 Databasen

Databasen til programmet, er jfr. afsnit 2.2, Datamodel, sammensat fra flere forskellige filer og en database. Da data er statiske, er indsættelse af data i basen lavet i et separat miljø.

Selve skemaoprettelsen er lavet i et sql script. Tabellerne der knytter sig til Trips er alle lavet med sql scripts, der henter og indsætter data fra week.sqlite tabellen, fra det oprindelige datasæt.

De tre "hovedtabeller", networknodes, edges og timetables, er alle lavet ud fra .csv filer, som er læst af et Python program, og herefter er de korrekte data indsat.

4.1.1 NetworkNodes

Data læses grundlæggende ud fra filen "network_nodes.csv", men der skal ske yderligere databehandling. Jfr. afsnit 2.5.1 om søgning på stationer, så skal der findes en metode til at gruppere stationerne, så der kan findes en nogenlunde logisk sammenhæng, mellem stationer der er placeret nært hinanden, og i brugernes øjne er det samme. I et 'rigtigt' transportsystem, ville dette nok være en delvis manuel proces, så man var 100% sikker på at få et korrekt resultat, men jeg vil forsøge at lave en algoritme.

Jeg antager derfor at hvis to stationer har 'næsten' samme navn og ligger tæt på hinanden, så er det samme station. Mht. navnet, så starter jeg med at fjerne info der beskriver om det er S-bahn, U-bahn eller bus/tram. Som eksempler henleder jeg til Tabel 2: Søgning på Alexander, her vil algoritmen fjerne bl.a. 'S+U' og distinktionerne i firkantede parenteser. Herefter sammenlignes de første tegn, og hvis de er ens, og stationerne er tæt på hinanden, er de i samme gruppe. Det fundne navn bruges ligeledes som 'shortStopName' som benyttes flere steder, bl.a. i den søgeliste som brugeren vælger stationer ud fra.

Efter en nøjere analyse finder jeg frem til, helt lavpraktisk ved at kigge på data, at to stationers første 13 tegn giver et rimeligt snit til at de har 'samme' navn. Jeg finder på samme måde frem til at 350 meter afstand giver et fornuftigt resultat, i forhold til at få dem grupperet korrekt. Dette er 100% ikke den korrekte løsning, men mit bedste bud. At gøre det 100 % korrekt, vil kræve et indgående stedkendskab.

Hvis to, eller flere, stationer får 'samme navn', tildeles de alle et ens unikt GroupId > 0, så det er muligt at fremsøge stationer i samme gruppe senere.

4.1.2 Edges og Timetables

Disse to tabeller bygges i Python scriptet "InsertIntoTimeTable.py". Her gennemløbes de rå data fra .csv filen "network_temporal_week.csv". Hver linje indlæses og det undersøges om der er registreret en edge mellem de to stop. Hvis ikke, tages næste ledige EdgeId, og indsættes i Edges tabellen. Til slut indsættes data om afgangen i Timtables tabellen med det nye, eller fundne EdgeId.

Slutteligt, skal Edges fodres med gå afstande mellem stationerne, som findes i "network_walk.csv". Afstanden er angivet i meter mellem to stop, men *kun* den ene vej. Jeg vælger at gemme afstanden i meter, da jeg så efterfølgende kan bestemme hvor hurtigt man går uden at skulle ændre på de oprindelige data. Desuden skal begge veje gemmes i datasættet.

4.1.3 Implementering af datamodellen i C#

Jeg benytter Entity Framework ver 6. EF med sqlite har ikke faciliteter til at ændre tabeller og migrationer som ved fx MySql eller SQLExpress, så al håndtering af databasen, opsætning af primær og fremmednøgler, skal håndteres manuelt. EF gør det muligt at lave forespørgsler i databasen med LINQ, og det er også muligt at tilgå tabeller via fremmed nøgler direkte.

Den overordnede klasse for datamodellen er i klassen *TransportSystem*. Ved at vælge Entity Framework, undgår jeg besværlig kode med connections og at skrive SQL direkte i min C# kode.

4.2 Beregning af afstand og tider

Noget af det meste basale i dette program er at programmet skal have en viden om hvor og hvornår det er.

Alle afstande er beregnet i meter, og alle tider er i sekunder. Det har en række fordele; Jeg slipper for omregninger internt, og tiderne i køreplanen er alle angivet i Unix tid[4], som tælles i sekunder. Det gør det også nemt at beregne forskellen mellem to tider, tallene trækkes bare fra hinanden. Jeg skal så kun omregne til kilometer/meter og timer/minutter, når resultaterne præsenteres for brugeren.

Stationernes placering af angivet ved alm længde/bredde (lon, lat) i grader. Der er flere måde at beregne afstand på, *haversine* benytter viden om at jorden er rund, eller mere simpel *Pythagoras.*

Haversine er meget nøjagtig, men mere beregningskrævende (benytter 7 trigonometri og 2 kvadratrods udregninger), mens Pythagoras er forholdsvis simpel at beregne. Jeg har udført en række test med Pythagoras og med de afstande der bruges i opgaven, er unøjagtigheden på få meter. Da der skal laves mange beregninger for hver søgning, vælger jeg Pythagoras:

$$x = \Delta\lambda * \cos\ \varphi_m$$

$$y = \Delta\varphi$$

$$d = R\sqrt{x^2 + y^2}$$

Udfoldet:

$$x = (lon_1 - lon_2) * \cos(\frac{lat_1 + lat_2}{2})$$

$$y = lat_1 - lat_2$$

$$R = 6.371.000$$

Der er lige den hage at latitude og longitude skal angives i radianer, så det skal omregnes inden, idet de i databasen er angivet i grader, men det er en simpel funktion: $rad = \deg * (\pi/180)$.

4.2.1 Gåafstande

Når man bevæger sig til fods, er der yderligere at tage højde for. Afstanden mellem de to stationer er angivet på forhånd fra filen network_walk.csv, og er indlæst i Edges tabellen. Når en gå strækning skal bruges, omregnes afstanden til tid. Men det at gå, udtrykkes jo ikke i sekunder, men hele minutter, så algoritmen omregner gåafstande til oprundede hele minutter, så selvom det tager 61 sekunder, vil resultatet blive 2 minutter.

Med denne viden, implementeres en klasse 'Distance' som håndterer alle forespørgsler i transportsystemet mht. tid og afstande. Klassen oprettes med et transportsystem og bruges som moder klasse efterfølgende.

4.2.2 Hjælpeklasse til tidsberegning

I den statiske klasse TimeHelpers, findes en række små metoder, som kan omregne tid. Mange af metoderne er lavet som *extensions* så de kan skrives efter hinanden med punktum mellem.

Der er metoder til at omregne Unix tid fra/til .net DateTime, og der findes metoder der kan omregne *nutids tid* til den tid der findes i køreplanen. Datasættet dækker en periode fra 25. april 2016 og en uge frem. Når en bruger angiver et tidspunkt, omregnes det til den samme dag/tid i den periode.

[4] Unix tid (også kendt som Epoc eller POSIX-time) tager udgangspunkt i 1/1-1970, midnat, og tæller sekunder gået derfra.

4.3 A* algoritmen

Jeg har benyttet følgende pseudokode fra wiki som grundlag for min implementering:
https://en.wikipedia.org/wiki/A*_search_algorithm#Pseudocode

```
function reconstruct_path(cameFrom, current)
    total_path := {current}
    while current in cameFrom.Keys:
        current := cameFrom[current]
        total_path.prepend(current)
    return total_path

// A* finds a path from start to goal.
// h is the heuristic function. h(n) estimates the cost to reach goal from node n.
function A_Star(start, goal, h)
    // The set of discovered nodes that may need to be (re-)expanded.
    // Initially, only the start node is known.
    // This is usually implemented as a min-heap or priority queue rather than a hash-set.
    openSet := {start}

    // For node n, cameFrom[n] is the node immediately preceding it on the cheapest path from start
    // to n currently known.
    cameFrom := an empty map

    // For node n, gScore[n] is the cost of the cheapest path from start to n currently known.
    gScore := map with default value of Infinity
    gScore[start] := 0

    // For node n, fScore[n] := gScore[n] + h(n). fScore[n] represents our current best guess as to
    // how short a path from start to finish can be if it goes through n.
    fScore := map with default value of Infinity
    fScore[start] := h(start)

    while openSet is not empty
        // This operation can occur in O(1) time if openSet is a min-heap or a priority queue
        current := the node in openSet having the lowest fScore[] value
        if current = goal
            return reconstruct_path(cameFrom, current)

        openSet.Remove(current)
        for each neighbor of current
            // d(current,neighbor) is the weight of the edge from current to neighbor
            // tentative_gScore is the distance from start to the neighbor through current
            tentative_gScore := gScore[current] + d(current, neighbor)
            if tentative_gScore < gScore[neighbor]
                // This path to neighbor is better than any previous one. Record it!
                cameFrom[neighbor] := current
                gScore[neighbor] := tentative_gScore
                fScore[neighbor] := gScore[neighbor] + h(neighbor)
                if neighbor not in openSet
                    openSet.add(neighbor)

    // Open set is empty but goal was never reached
    return failure
```

*Figur 6: Pseudokode, A**

Algoritmen tager ikke højde for at der arbejdes i en dynamisk graf, så der er nogle ændringer som skal tage højde for at der er ventetid i transportsystemet.

A* algoritmen implementeres som en selvstændig klasse, som dog nedarver fra 'Distance' klassen. På den måde har den adgang til alle metoder vedr. afstande og tids beregning, som senere skal bruge i beregning af heuristik.

4.3.1 Input til A*

A* klassen institueres, med EF-klassen, TransportSystem.
Metoden FindPath kaldes med følgende parametre:

- StopId på afgangsstationen
- StopId på ankomststationen

- Tidspunkt for afgang eller ankomst
- Retningen for søgningen (er ovennævnte tidspunkt afgangs- eller ankomsttid).

4.3.2 Output fra A*

A* afleverer en simpel liste af klassen PathItem. Et PathItem består kun af to heltal, et EdgeId og et TimeTableId. Disse data er nok til at en anden klasse kan skabe den fulde rute information for brugeren. PathItem, er således et *Data Transfer Object* hvilket hjælper med at afkoble A*'s interface fra den underliggende datastruktur i TransportSystem klassen.

4.3.3 Datastrukturer

A* benytter sig af en minimums prioritets kø, i algoritmen kaldt *openSet* (linie 14). I denne opgave er der brugt et plugin "High Speed Priority Queue"[5] som er specielt udviklet til denne type opgave. High Speed Priority Queue, benytter en implementering med et array af fast længde.

Køen indeholder en klasse *QueueNode* (nedarvet fra *FastPriorityQueueNode*) som indeholder StopId, tid og TripId. Denne implementering gør, at fScore bliver overflødig. Køens prioritet bygger på den *forventede* ankomsttid til slut destinationen, mens indholdet siger noget om hvor algoritmen står i grafen, og præcist hvilket tidspunkt.

cameFrom (18) er implementeret som et Dictionary, hvor nøglen er StopId, og værdien er en intern klasse *TracebackNode*, der holder den information som netop er nødvendig for at kunne bygge ruten, hvis en sådan findes. Man skal huske på, at når ruten skal bygges, foregår det baglæns.

gScore (22) er endnu et Dictionay, igen er nøglen StopId, hvor værdien er den tidligste fundene ankomsttid til dette stop fra vores udgangspunkt.

4.3.4 Beskrivelse af algoritmen

Algoritmen kører en løkke, hvor der startes med at tage en node af prioritetskøen, indtil køen er tom, eller der er fundet en vej gennem grafen (linje 29).

Enten er den node der er taget fra køen, ankomststationen eller også skal alle naboer til noden findes. Alle naboer findes vha. en søgning i Edges tabellen:

```
// get the neighbours to the currentNode
var neighbours = ts.Edges
                   .Where(e => e.FromStop == currentNode.StopId)
                   .Select(e => new { e.EdgeId, e.ToStop, e.DistanceWalk });
```

Denne søgning returner en liste af noder, som der kan itereres over. Her vælger jeg at bygge listen kun over det data som er nødvendig. Jeg kunne således godt have ladet søgningen returnere en liste af Edges, men alle Edges i listen ville have det samme FromStop, hvilket er redundant data. Jeg vil som udgangspunkt *kun* have data i mine algoritmer som jeg har brug for.

Inner løkken, som undersøger alle naboer, kan nu gå to veje:

Enten at gå eller køre til næste node, og det er bestemt ud fra DistanceWalk > 0. Skulle det ske, at den tid det tager at gå, er længere en det fastsatte maksimum gå tid springes til næste node.

Maximum tiden er indsat, idet datasættet indeholder gåafstande mellem de af noderne der er mindre en 1 km væk. Hvis jeg ikke indsætter denne betingelse, vil algoritmen i de nogle tilfælde vælge at starte med at gå fra stop til stop, indtil den bliver indhentet af bussen/toget. Det er selvfølgelig også en løsning og sådan set ikke forkert, men ikke det en bruger forventer. Men der findes specielle tilfælde, fx om aftenen, hvor

[5] https://github.com/BlueRaja/High-Speed-Priority-Queue-for-C-Sharp

der ikke kører så mange busser. Her kan det nogle gange betale sig at gå til den nærmeste tog station, i stedet for at vente på bussen.

Desuden testes der for at man ikke kan gå to strækninger i træk, det giver ingen mening for brugeren. Det er dog muligt at gå en strækning midt i ruten, for det skal være muligt at skifte, og det indbefatter ved nogle af de større stationer, at man skal gå, hvis der kun er ét stopId.

I det andet tilfælde søges nu på hvornår den næste afgang er fra noden.

I gScore findes den tidligste ankomsttid, og næste afgang er så strækningen som har det samme EdgeId i TimeTables tabellen. Jeg søger den næste afgang som er lig med eller større end tiden i gScore. Her er det værd at bemærke at alle tider i køreplanen er angivet som hele minutter, så et tog ankommer og kan afgå på *samme* minuttal! Dette løses i den lokale metode GetNextDeparture:

```
TimeTable GetNextDeparture(int edgeId, long departureTime)
{
    return ts.TimeTables
       .Where(t => t.EdgeId == edgeId && (t.DepartureTime >= departureTime))
       .OrderBy(t => t.DepartureTime)
       .FirstOrDefault();
}
```

Hvis denne søgning ikke giver et resultat, findes ingen forbindelse, og der springes videre til næste nabo.

Hvis der skiftes transportmiddel, og der ikke ligger en gåtur mellem, skal der indføres en skiftetid. I første omgang er der fundet en afgangstid, som beskrevet tidligere. Der indføres herefter en test på om der er skiftetid nok, hvis et skift har fundet sted, og man ikke er gået dertil.

```
if ((currentNode.TripId != -1)
    & (currentNode.TripId != timeTableEntry.TripId)
    & (timeTableEntry.DepartureTime < gScoreTemp + DefaultTimeToChangeTransport))
```

- currentNode.TripId er lig -1 hvis sidste ben i ruten var på gåben.
- currentNode.TripId er lig med den nyligt hentede post i TimeTable tabellen hvis sidste ben på ruten foregik med samme rute
- Slutteligt testes om en evt. skiftetid er overholdt.

Er alle disse tests ikke overholdt, lægges skiftetiden til, og en ny søgning laves igen.

Til sidst i løkken undersøges på om resultatet er hurtigere end et allerede fundet resultat:

```
if (arrivalTime < gScore.TryGetValueOrMax(n.ToStop))  // quicker way found
```

Den fundne strækning indsættes i cameFrom, for at kunne finde vejen tilbage

```
cameFrom[n.ToStop] = new TraceBackNode { ToStop = currentNode.StopId, EdgeId = n.EdgeId
, TimeTableId = timeTableId};
```

og der oprettes et objekt til prioritetskøen.
Hvis dette objekt ikke allerede findes i køen, indsættes det med den udregnede heuristik.

```
if (!heap.Contains(qn))
{
    int heuristik = DistanceTime(n.ToStop, b);  // estimated traveltime from this node
    heap.Enqueue(qn, heuristik + arrivalTime);
}
```

4.3.5 A* baglæns

Når der skal laves en baglæns søgning, dvs. ud fra en ønsket ankomsttid, så forgår det i metoden FindePathReverse. Algoritmen er langt hen ad vejen den samme som i forlæns søgning, dog er afgangstid erstattet af ankomsttid i sammenligninger.

Heuristikken til baglæns ser ud som følger: $f(v) = h\,(v,d) - \ cost\ (v)$, så der laves en lille ændring i kaldet til prioritetskøen:

```
heap.Enqueue(qn, heuristik - departureTime);
```

4.3.6 Beregning af heuristik (optimal hastighed)

Iflg. teorien og mit tidligere afsnit om heuristik, så findes den optimale heuristik ved jfr. Figur 5: Amit Patel, Standford, ved at sætte heuristikken til at være *nøjagtig* prisen for at komme fra a til b.

Heuristikken beskrives som funktionen $f(v) = cost\ (v) + h\ (v,d)$. v er den node vi er kommet til, og d er destinationen. cost (v) er den udregnede tid der er brugt til at komme fra start til v. $h\ (v,d).)$ skal derfor udtrykke tiden det tager at komme af v til d.

I klassen distance findes en metode der på baggrund af hastighed udregner transporttiden mellem to stop, og det er netop $h\ (v,d)$. Men den metode skal have en viden om hvilken hastighed vi bevæger os med, og hvordan findes den optimale hastighed i en dynamisk graf?

Udfordringen er, at der er ventetider i grafen, når der skal skiftes transportmiddel og tog eller bus kører med forskellige hastigheder, så vi ved det faktisk ikke, så det må bero på bedste gæt, og nogle praktiske forsøg.

Iflg. Tabel 1: Hastigheder i Berlin transportsystem, så er gennemsnits hastigheden for et tog ca. 38 km/h, for U-bahn 24 km/h og for bus 20 km/h. Det kunne derfor være et udgangspunkt at lægge hastigheden et sted mellem 38 og 24 km/t. Men er det rigtigt, og giver det korrekte søgninger?

4.3.6.1 Praktisk undersøgelse

For at undersøge i praksis, har jeg konstrueret et program som tester A* ved at ændre hastigheden i heuristikken. Programmet laver et antal tilfældige søgninger fra a til b på et tilfældigt tidspunkt. For hver søgning, startes med en A* hvor $h\ (v,d) = 0$, dvs. Dijkstra's algoritme. Denne søgning bruges som reference til de senere søgninger, hvor efter hastigheden i h gradvist sættes op. Kun de søgninger som har samme resultat som Dijkstra godkendes og tæller med i statistikken. Programmet finder desuden for hver søgning, den laveste hastighed som giver valide resultater, og den hastighed med validt resultat hvor søgningen går hurtigst. Resultaterne gemmes i en komma separeret fil, og behandles i Excel.

Resultater ved 500 tilfældige søgninger, hvoraf 458 gav resultat[6]:

	Dijkstra	4 m/s	5 m/s	6 m/s	7 m/s	8 m/s	9 m/s	10 m/s	11 m/s	12 m/s
Gennemsnitstid (sek.)	4,7	1,7	1,7	1,9	2,2	2,5	2,8	3,0	3,2	3,4
Største tid (sek.)	14,1	13,9	12,8	12,5	12,0	11,4	11,8	11,8	12,0	12,5
Antal korrekte	458	233	304	349	386	387	398	402	400	405
% korrekte	100%	59%	71%	78%	84%	86%	85%	86%	85%	88%
% tidsforbrug i forhold til Dijkstra	---	36%	37%	42%	48%	53%	59%	64%	69%	72%

Alle tiderne i tabellen ,er beregningstider.
Hastigheder under 4 m/s giver ikke særlig mange valide resultater, og derfor ikke medtaget.
Ovenstående tabel viser, at allerede ved 4 m/s er heuristikken valid for over 50 % af turene.
Hastigheden 38 – 24 km/h svarer til 6,6 - 10,5 m/s.

Af undersøgelsen ses at det ikke er trivielt at finde en optimal heuristik, som i alle tilfælde giver et korrekt resultat, men at en hastighed på 7 m/s giver overvejende valide resultater, med et tidsforbrug på ca. 50 % af en ren Dijkstra.

[6] Datamaterialet findes i Bilag 2

En nærmere undersøgelse af data for heuristikken, viser også, at nogle ruter dømmes ugyldige, selvom de har samme transporttid. Dette skyldes muligvis at min test algoritme kun godtager den vej som Dijkstra finder som korrekt, man at der ved nogle tilfælde findes en anden, og lige så hurtig vej.

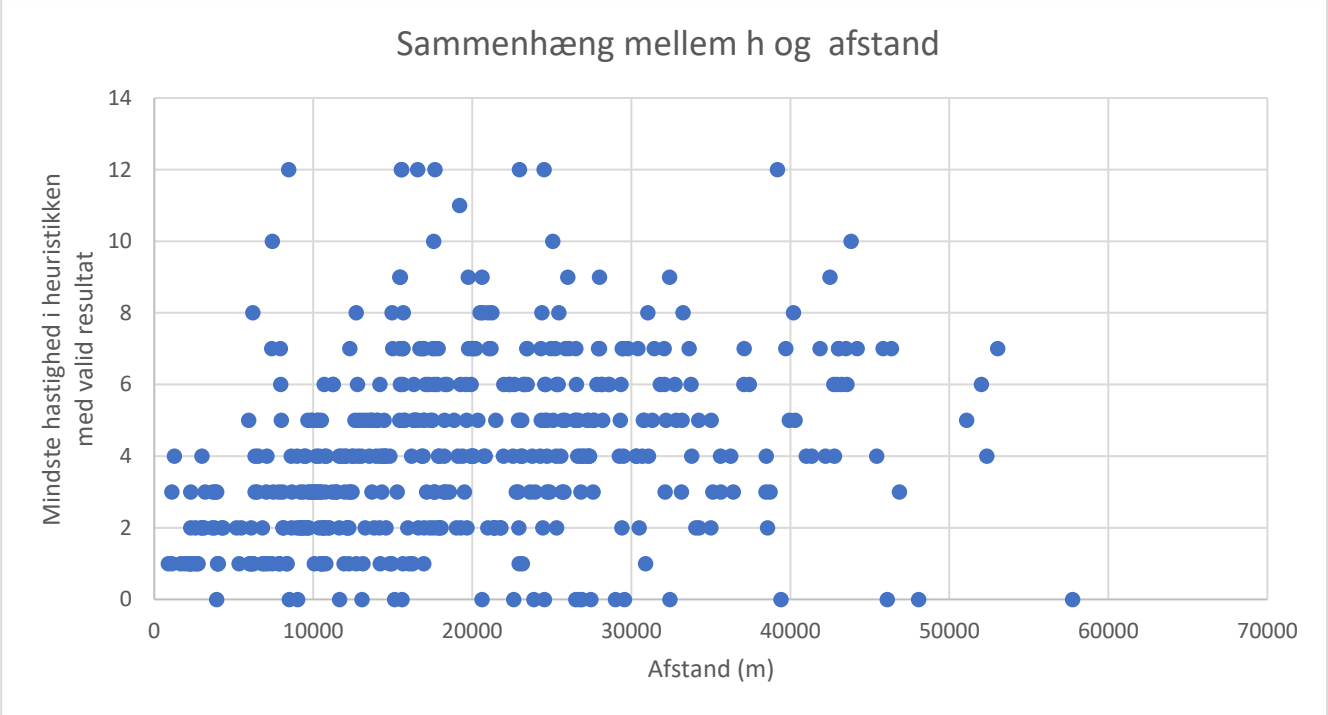
Denne tabel viser sammenhængen mellem heuristikken og rute tid, når vi går ud fra at den tid som Dijkstra har fundet, er den optimale rute tid:

	4 m/s	5 m/s	6 m/s	7 m/s	8 m/s	9 m/s	10 m/s
Antal der har samme tid	335	382	412	420	430	426	432
% med samme tid	73%	83%	90 %	92 %	94 %	93 %	94%
Antal med forskellig tid	124	77	47	39	29	33	27
Gennemsnit ekstra tid	0:12	0:11	0:08	0:05	0:08	0:05	0:07

Ud fra tabellen ses det at 7 m/s giver et udmærket kompromis mellem antal rigtige, og den ekstra tid som ruten tager, sammenholdt med tidsforbruget i algoritmen.

4.3.6.2 Sammenhæng mellem heuristik og afstand

Jeg havde også en tese om at heuristikken muligvis kunne afhænge af afstanden. Kunne det være således, at jo længere der var, jo hurtigere skulle heuristikken bevæge sig? Ved at sætte afstand op mod den mindste hastighed hvormed der findes en korrekt vej fås følgende diagram:



Figur 7 Sammenhæng mellem h og afstand

Ved at kigge på diagrammet, ses at på korte afstande kunne hastigheden muligvis vælges mindre, for på den måde at optimere A*. Det kunne derfor være en mulighed at vælge at hastigheden sættes til 4 m/s for afstande under 10 km, og for 7 m/s over 10 km. Der er dog marginale forbedringer i tidsforbruget af A* så jeg vælger ikke at ændre og komplicere algoritmen yderligere.

4.3.7 Store ventetider i transportsystemet

Hvis næste afgang er uforholdsmæssig lang tid væk, fx næste dag, besøges alle mulige andre veje, førend den direkte vej findes, men i praksis tager ingen afsted for at vente 5 timer ved et stop. Derfor undersøgte jeg om der skulle laves en test som dropper en forbindelse hvis ventetiden bliver for lang (fx mere end 2 timer)

Ved at udføre en række forsøg mht. ventetid til næste transportmiddel, viser det sig at være af marginal betydning. De noder hvor ventetiden er lang, ender alligevel bagest i prioritetskøen, og i grelle tilfælde sker der det, at A* tager længere tid om at finde vej.

En mulighed er i stedet forsøge at indføre et check inden A*, om der overhovedet er afgange fra afgangs stoppet, inden søgningen starter. Hermed kan jeg gøre brugeren opmærksom på, at der måske skal vælges et andet tidspunkt. Men det er dog ikke uden omkostninger. Tag eksemplet, hvor man tager et stoppested tæt ved en togstation: her vil en bruger måske være tilbøjelig til at gå 5 minutter til nærmeste station, for at tage toget, frem for at vente lang tid på den næste bus.

Det skal bemærkes, at der i nogle områder er meget kort afstand mellem togstationer, mens der i yderområderne af Berlin er længere afstand. Dette kunne måske løses ved at kigge på afstanden til nærmeste station, og dermed beslutte om man vil bede algoritmen om at gå efter første stop eller ej, men i så fald, så skal vi have en viden om hvor tæt man er til en togstation, og den viden har vi ikke umiddelbart, så det fravælges her, med andre ord, jeg dropper løsningen.

4.3.8 Indeks i databasen

A* laver en række søgning i databasen og disse kan optimeres. Den nemmeste metode, er at sørge for at der er lavet indeks i databasen som gør søgningerne mere effektive.

Et indeks skal laves så det understøtter de søgninger der laves. Derudover skal der laves indeks til fremmed nøgler, det gør sqlite ikke automatisk.

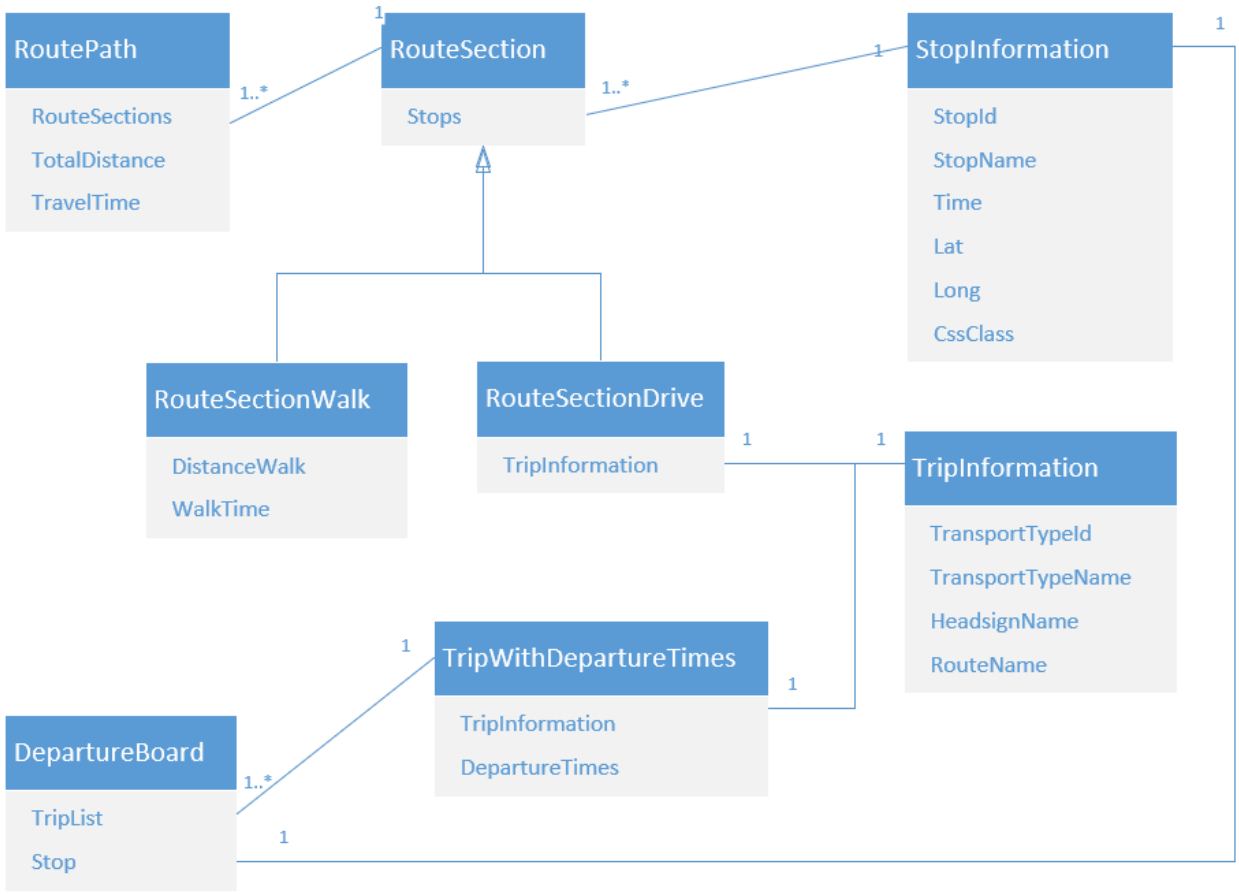
Fx er der lavet et indeks, som letter søgningen på EdgeId og ArrivalTime i TimeTables tabellen.

4.4 Viderebehandling af data efter A*

A* afleverer en liste af PathItem, som er en liste med to heltal, EdgeId og TimeTableId. Denne liste, hvis ikke den er tom (!), skal oversættes til en *ViewModel* som kan sendes til browseren i front-end.

En ViewModel er en datastruktur, som præsenterer nøjagtigt de data som browseren har brug for. Denne datastruktur bør være afkoblet af programmets interne datastruktur, for bl.a. af imødegå hacking. Nu er denne applikation en read-only, men i tilfælde hvor der skal skrives data i en database, er det en fordel ikke at præsentere den underliggende datamodel.

Den datamodel som er bygget til formålet at finde korteste vej, egner sig meget dårligt til at præsentere data til en bruger, og den model som skal bruges til at beskrive en rute, er beskrevet her.



Figur 8 ViewModel

4.4.1 Viewmodel til strækning (RoutePath)

Af Figur 8 ViewModel, ses datastrukturen for den viewmodel der bruges til at præsentere den fundne rute til brugeren.

En rute er en liste af strækninger (RouteSections). Hver strækning er et stykke af den samlede rute, som brugeren kører i det samme transportmiddel eller går mellem to stationer. En strækning er også en liste af stops man kører/går mellem. Hvis der køres, er der ligeledes en strækningsinformation tilknyttet (TripInformation), som fortæller hvilken rutebetegnelse (fx bus nr.) navnet på endestationen og hvilket slags transportmiddel der er tale om. Hvis der er tale om en gå strækning, er afstanden og tiden angivet. For hvert stop, er navnet, hvilket tidspunkt man starter eller stopper på stationen etc. angivet. Det er værd

at bemærke, at alle stop, undtagen det sidste i listen, er afgangstider, mens det sidste er en ankomsttid. Alle tider er angivet i Unix tid, så det er op til web-front end, at konvertere og angive disse korrekt. Ligeledes er den samlede rejsetid også angivet i sekunder

4.4.2 Viewmodel til afgangstavler (DepartureBoard)

En afgangstavle er en beskrivelse af alle forbindelser fra et bestemt stoppested. Når en bruger vælger at få præsenteret en tavle, vises de næste 5 afgange indenfor de næste 2 timer.

Strukturen er vist i **Fejl! Henvisningskilde ikke fundet.**. En afgangstavle (DepartureBoard) er foruden o plysninger om selve stoppestedet (StopInformation), en liste af ture med deres strækningsinformation (TripInformation) og en simpel liste af tidspunkter, for hvornår denne forbindelse er. Det ses at der er overlap mellem de to viewmodels, da nogle datastrukturer kan bruges på tværs.

4.5 Web front-end implementering

Til visning af data i web front-end benyttes en række plug-ins i Java script, disse beskrives her.

4.5.1 Twitter typeahead

Dette plugin bruges på index siden, til at støtte brugeren med at indtaste navne på stationer. Når brugeren således har tastet 2 tegn, vises en drop-down liste, med forslag der passer på de indtastede tegn. Her benyttes ShortStopName, som tidligere beskrevet i afsnittet om NetworkNodes. Typeahead benytter et kald til et api som returnerer en liste af stationer som matcher den forespørgsel som brugeren har indtastet.

Dette plugin er dog ikke ufejlbarligt, jeg har opdaget at der er flere tilfælde hvor det ikke viser den korrekte liste af mulige navne. Jeg har tjekket med PostMan[7] om mit api svarer korrekt, og det er tilfældet, men af søgninger på internettet, kan jeg se at andre bruger har samme oplevelse. Det er også længe siden at der sidst har været en opdatering, og i mellemtiden er der lavet en branch af koden i en anden kontext, men jeg har alligevel valgt at holde mig til Twitters ældre udgave, da dette er en mindre detalje i mit projekt.

4.5.2 Kort

Jeg har valgt at vise ruten på et kort, det er også derfor at min View model indeholder latitude/longitude informationer om hvert enkelt stop på ruten. Der findes flere plugins som kan tegne på et kort, jeg har bl.a. undersøgt openlayers.org som viste at være meget avanceret men endte med at benytte leafletjs, som på meget simpel vis, kan tegne ruten. Dette foregår i JavaScript ud fra de data der er i RoutePath View Model.

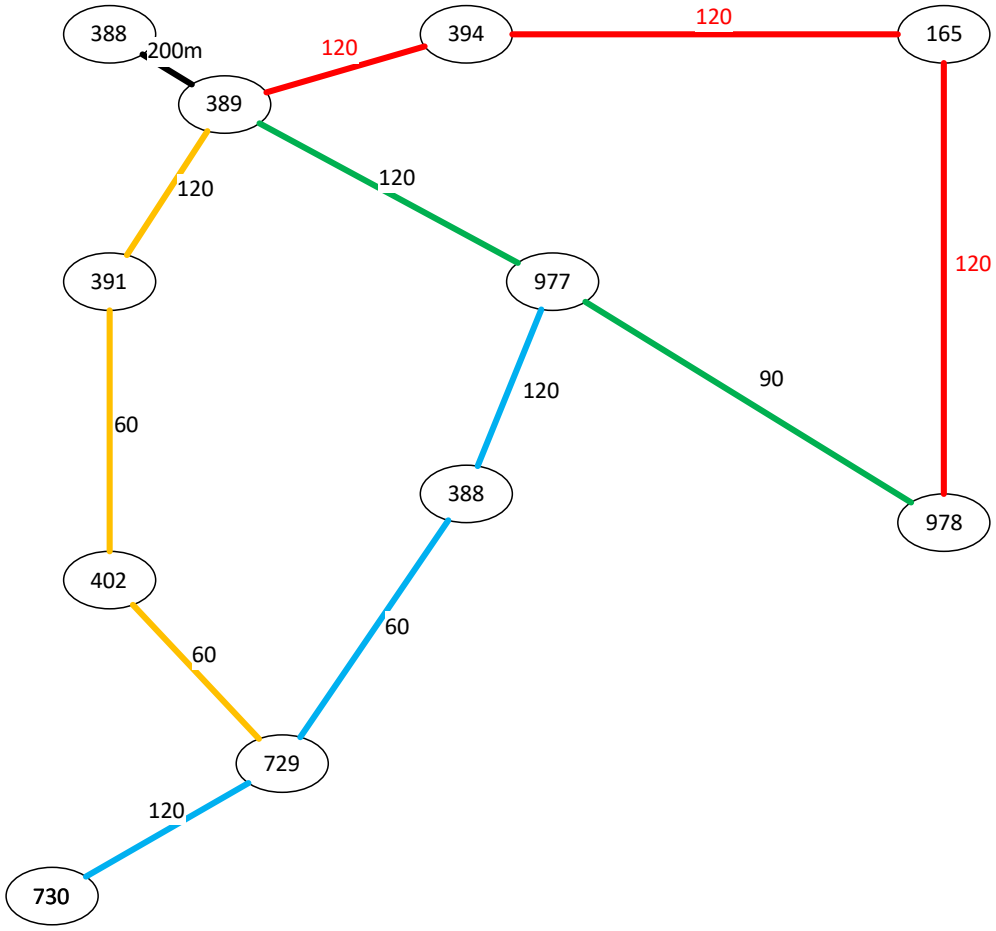
4.5.3 API

Som tidligere beskrevet er der lavet et lille Api, som giver en liste af stationsnavne, ud fra en simpel forespørgsel. Det er en simpel søgning på begyndelsesbogstaver, renset for store eller små bogstaver.

[7] PostMan er et værktøj til at lave bl.a. forespørgsler på et Api (Application Programmers Interface)

5 Test af A stjerne

For at teste A* algoritmen har jeg lavet et lille datasæt, men nogle få stationer og liner.



Figur 9 Test datasæt

Sammen med datasættet har jeg lavet en simpel køreplan, som jeg har brugt til at sikre at A* og andre algoritmer har virket korrekt. Det er ikke muligt at lave en ordentlig test på det store datasæt, da det er alt for uoverskueligt. Mit testdatasæt opererer desuden med en tidsregning der starter på tid 0, som gør det nemmere at se tider, da tallene er små, sammenlignet med Unix tiderne i det store datasæt.

Testdatasættet er gemt i en separat sqlite database. Jeg har lavet en simple konsol applikation i A-star projektet, hvor jeg enkelt kan ændre om jeg kører på test, eller det store datasæt. I den applikation, indtastes stationsnavne kun med ID nummer, og ruten vises på samme måde som i web applikationen, der er lavet en simple ToString metode som printer ViewModels i konsollen.

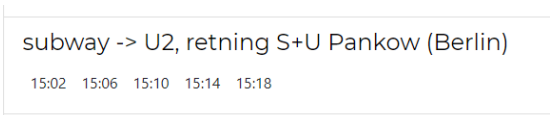
Testdata køreplanen og EdgeId tabellen er vedlagt som bilag 3.

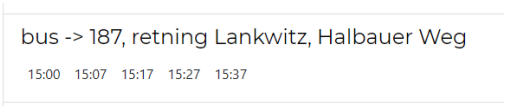
6 Resultater af søgninger

Ved at kigge på en række søgninger, finder jeg, at selvom jeg har fundet den optimale hastighed for heuristikken, giver min algoritme alligevel ikke altid korrekte resultater!

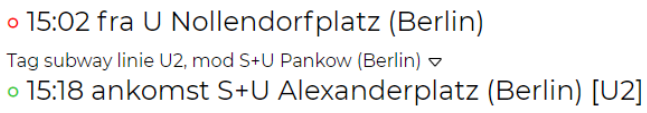
Fx vil søgningen fra Nollendorf platz til Alexander Platz kl. 15:00 en søndag, vise følgende rute:

Ved at kigge nærmere på ruten, så skal algoritmen bruge en forbindelse mellem Nollendorf og Bülow Strasse. Den første ledige er kl. 15:00 med en bus, mens den korrekte med U-bahn er kl. 15:02. Men den vælges bare ikke, da algoritmen vælger den første og bruger den.





Her er den korrekte rute:



Dette er faktisk kun et problem ved de af stationerne, som kun har ét stopId, men flere forbindelser, som fx Nollendorf. Omvendt har Alexander Platz ikke problemet, for her har hver rute sit eget stopId, og de enkelte stopId er forbundet med gå ruter.

Derfor skulle grafen ændres, således at de steder hvor der er flere forbindelser mellem to stop, skal der også være to kanter, ligesom det er lavet med gå ruterne.

På den måde vil alle linjeforbinder blive undersøgt, når A* kigger på naboer til noden, og algoritmen er sikker på at få kigget på alle mulige forbindelser, og ikke kun den første, som muligvis ikke var den hurtigste. Den viden har mit oprindelige data bare ikke.

7 Konklusion

Det viser sig, at ved valg af en optimal hastighed for heuristikken, kan det lade sig gøre at finde den hurtigste rute i ca. 92 % af tilfældene med et halveret tidsforbrug i forhold til Dijkstra. De resterende 8% er lidt forkerte, men gennemsnitlig kun en anelse længere eller kortere. Jeg har i denne opgave ikke analyseret, hvad som går galt med de sidste 8 %.

Men det viser sig også at der er nogle tilfælde hvor algoritmen går galt, og hvor ikke alle mulige forbindelser undersøges. Dette er naturligvis et problem, og bør ændres i en fremtidig version af programmet. Heldigvis ser det ud til at det kun er en ændring af tabellen med Edges der skal ændres. Desværre så er det ikke muligt entydigt at skabe de data ud fra det oprindelige datasæt, her skal man nok helt tilbage til det oprindelige GTFS-feed.

Når det så er sagt, så kan applikationen præsentere en rute for en bruger, ud fra de afgangs og ankomststop som er angivet og det er muligt for en bruger at vælge et stationsnavn, som giver mening.

Brugen af Bootstrap, muliggør desuden at browser side ser fornuftig ud, ved forskellige formater.

8 Perspektivering

Hvis dette program skulle have en fremtid og være relevant, så skal det for det første lave korrekte søgninger, og lige så vigtigt, læse et live GTFS-feed. Desuden skal det selvfølgelig ligge på en web-server som er offentlig tilgængelig.

En funktion som ville mangle i en app som denne, er også muligheden for at se hvordan en rute kører eller se en stoppestedstavle. Det er fx ikke muligt at søge oplysninger om hvilke stop bus 106 har, hvornår den kører osv. Informationer som en bruger vil forvente at kunne få. Ligeledes skulle der være en funktion som finder nærmeste stoppested ud fra brugerens GPS-koordinater.

9 Links til repository

Opgaven ligger på GitHub. Der er ingen sqlite database. Dette er pga. begrænsninger i filstørrelsen som GitHub har til ikke betalende brugere.

Link: https://github.com/jensgauldk/TransportSystem

10 Litteraturliste

Merrifield, T. (2010). Heuristic Route Search in Public Transportation Networks. Chigago, Illinois, USA.

Patel, A. (n.d.). *Heuristics*. Retrieved from Red Blob Games: http://theory.stanford.edu/~amitp/GameProgramming/Heuristics.html#a-stars-use-of-the-heuristic

11 Bilag

Bilag 1: A collection of public transport network data sets for 25 cities
Bilag 2: Excel ark, med data brugt i afsnit 4.3.6.1
Bilag 3: Test data