# Nooit meer een briefje halen

Het maken van een dynamisch navigatiesysteem binnen het schoolgebouw van het Hyperion Lyceum

Tobias Crommelin en Dylan Spence

Begeleider: Hugo Schouten

Wiskunde D & Informatica

Verslag Profielwerkstuk VWO 6

Hyperion Lyceum, Amsterdam

5 februari 2025

# Abstract

Dit onderzoek betreft het maken van een navigatiesysteem binnen het schoolgebouw van het Hyperion Lyceum. Door de snelste routes aan te geven verbetert het systeem de punctualiteit van de leerling en geeft het houvast voor bezoekers die onbekend zijn met het gebouw. Het systeem is gemaakt door het schoolgebouw te modelleren als een graaf en een programma te schrijven dat de kortste routes hierop vindt door middel van Dijkstra's kortste pad algoritme. Ook is een wiskundig model ontwikkeld om voetgangersstromen te simuleren en files te voorspellen. Dit is meegenomen in het berekenen van de kortste route. Al het ontwikkelde materiaal is samengebracht in een website die gebruikt kan worden door leerlingen en bezoekers om de snelste route te berekenen tussen twee plekken op school.

# Inhoudsopgave

**Abstract** 2
**Inhoudsopgave** 3
**Introductie** 5
**Symbolenlijst** 6
**1 Theoretisch kader** 8
1.1 Grafentheorie 8
1.1.1 De Graaf 8
1.1.2 De Gerichte Graaf 8
1.1.3 De Gewogen Graaf 9
1.1.4 Matrixvoorstelling en adjacency list 9
1.2 Dijkstra’s algoritme 11
1.3 Stromen 13
1.3.1 s,t-stroom 13
1.3.2 Nash-stroom en optimale stroom 15
1.4 Verkeerstheorie 17
1.4.1 Fundamenteel verband 17
1.4.2 Fundamentele diagrammen 18
1.4.3 Voetgangersstromen 19
**2 Graaf Hyperion Lyceum** 21
2.1 Verantwoording 21
2.1.1 Relevante locaties 21
2.1.2 Simpliciteit versus granulariteit 21
2.1.3 De glijbaan 24
2.2 De graaf 24
2.3 Gewichten 24
2.3.1 Deuren openen 24
2.3.2 Trappen lopen 25
2.3.3 Bijzondere gewichten 26
2.3.4 Loopsnelheden 26
2.4 De gewogen graaf 27
**3 Kortste pad programma** 28
3.1 Bestandsbeheer graaf 28
3.2 Code Dijkstra’s algoritme 28
**4 Casus s,t-stroom** 30
4.1 Vertragingsfunctie 30
4.2 Stromingsmodel 31
4.2.1 De deelgraaf 31
4.2.2 Verantwoording 32
4.2.3 Codering 32
4.3 Resultaten 34
4.4 Uitbreidmogelijkheden en tekortkomingen 36
**5 Stroommodel** 37

5.1 Dubbel Gewogen Graaf........37
5.2 Code........37
5.3 Resultaten........40
5.4 Toepassing........40
5.4.1 Van lokaal naar lokaal........41
5.4.2 Tussen les en pauze........42
5.4.3 Start van de dag, 9:00........43
5.4.4 Programmering in applicatie........44
**6 Gebruikersinterface (GUI)........47**
6.1 De Flask code........47
6.2 Visualiseren graaf met JavaScript........49
6.3 Responsive design........50
6.4 Mappenstructuur en hosting........51
**7 Conclusie........52**
7.1 Hoogtepunten........52
7.2 Ruimtes voor verbetering........53
**8 Discussie........54**
8.1 Testen systeem........54
8.1.1 Steekproefmethodiek........54
8.1.2 Uitvoer en resultaten........55
8.2 Testen verkeersmodel........56
8.2.1 Discussiepunten........56
8.2.2 Verificatiemethoden........57
**Bijlage A: Alternatieve aanpak van het dilemma tussen granulariteit en simpliciteit...58**
**Bijlage B: De (ongewogen) graaf van het Hyperion Lyceum........63**
**Bijlage C: De gewogen graaf van het Hyperion Lyceum........66**
**Bijlage D: De graaf met breedtes van het Hyperion Lyceum........69**
**Bijlage E: Externe links........72**
**Bijlage F: impressie gebruikerservaring........73**
**Bijlage G: logboek........77**
**Literatuurlijst........81**

# Introductie

*"Please get a note…"* zei de Engels docent altijd wanneer je, ook maar een seconde na het begin van de les, het lokaal binnenliep. En zo was je alweer vroeg in de ochtenddauw fietsend onderweg om je om kwart over acht op school te melden. Dit soort situaties, waarbij leerlingen te laat komen, zijn frustrerend. Niet alleen voor de leerling, maar ook voor de leraar, wiens les wordt verstoord door de laatkomer. Toch is het makkelijk te voorkomen, door op tijd te vertrekken en vervolgens de snelste route naar het klaslokaal te nemen. Echter, het is niet altijd gemakkelijk te bedenken wat de snelste route is, vooral in onze school - het Hyperion Lyceum - met haar "labyrintstructuur". En hoe weten leerlingen wanneer ze moeten vertrekken?

Dit onderzoek richt zich op het verbeteren van de punctualiteit van leerlingen door een navigatiesysteem te ontwerpen voor binnen het Hyperion Lyceum. Ook helpt het mensen die nog onbekend zijn met het gebouw om de weg te vinden. Dit systeem vertelt gebruikers welke route ze moeten nemen en wanneer ze moeten vertrekken. Zo komen ze op tijd in de les en hoeven ze geen briefje te halen. Het idee is geïnspireerd door navigatiesystemen zoals Google Maps en TomTom. Dit onderzoek tracht een vergelijkbaar systeem te creëren dat specifiek is afgestemd op onze schoolomgeving.

De hoofdvraag van dit onderzoek luidt: "Hoe kan een navigatiesysteem ontworpen worden voor het schoolgebouw van het Hyperion Lyceum?" Het onderzoek is verdeeld in vier deelvragen:

- Hoe functioneren grafen en hoe werkt Dijkstra's algoritme?
- Hoe kan het Hyperion Lyceum voorgesteld worden als een gewogen graaf?
- Hoe kan de beweging van mensenmassa's meegenomen worden in het vinden van het kortste pad?
- Hoe kan een gebruikersinterface ontworpen worden, voor leerlingen en bezoekers?

Het onderzoek is opgebouwd uit acht delen. In het *Theoretisch kader* worden de basisprincipes van grafentheorie en verkeerstheorie uiteengezet, waaronder een uitleg van Dijkstra's algoritme en de dynamiek van voetgangersverkeer. Het deel *Graaf Hyperion Lyceum* beschrijft de methodiek voor het omzetten van de structuur van het schoolgebouw in een graaf. Vervolgens richt het hoofdstuk *Kortste pad programma* zich op het schrijven van een programma dat het kortste pad vindt aan de hand van Dijkstra's kortste pad algoritme. Deze wordt toegepast op de eerder gemaakte graaf van de school. In het deel *Casus s,t-stroom* wordt een casus behandeld over de deelvraag "Hoe nemen we de beweging van mensenmassa's mee in het vinden van het kortste pad?". In dit hoofdstuk wordt er code geschreven om een leerlingenstroom tussen twee lokalen te simuleren. Dit wordt als opstap gebruikt voor het volgende hoofdstuk: *Stroommodel.* Hierin wordt het systeem dat gemaakt is in de casus door de gehele school toegepast. Om de ontwikkelde code ook bruikbaar te maken voor leerlingen en bezoekers van het Hyperion Lyceum, wordt een *Gebruikersinterface* ontwikkeld. Dit wordt besproken in het gelijknamige hoofdstuk. In de *Conclusie* wordt er teruggeblikt op het onderzoek en zijn verschillende fasen van dit onderzoek. Tot slot worden in de *Discussie* de resultaten van het systeem getest, sterke punten en verbeterpunten besproken, en aanbevelingen gedaan voor toekomstige verbeteringen.

# Symbolenlijst

| Symbool | Betekenis |
|---|---|
| $A$ | Verzameling gerichte zijden |
| $D$ | Gerichte graaf |
| $dt_{gem}$ | Gemiddelde overschatting op een dilemmaplek |
| $e$ | Zijde (Edge in Engels) |
| $E$ | Verzameling zijden |
| $f$ | Stroom |
| $f_{nash}$ | Nash-stroom |
| $f_P$ | Deelstroom |
| $f^*$ | Optimale stroom |
| $F$ | Verzameling mogelijke stromen |
| $G$ | Graaf |
| $k$ | Dichtheid stroom (ééndimensionaal) |
| $k'$ | Dichtheid stroom (tweedimensionaal) |
| $k'_{jam}$ | Maximale dichtheid |
| $k_{krit}$ | Kritieke dichtheid |
| $L$ | Vertraging |
| $n$ | Aantal voertuigen in een stroom |
| $P$ | Pad |
| $\mathcal{P}$ | Verzameling paden |
| $PoA$ | Prijs van anarchie |
| $q$ | Intensiteit stroom (ééndimensionaal) |
| $q'$ | Intensiteit stroom (tweedimensionaal) |

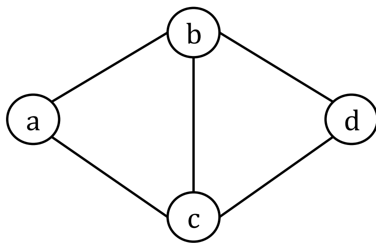
| | |
|---|---|
| $q_{max}$ | Maximale intensiteit |
| $\mathbb{R}$ | Verzameling reële getallen |
| $SC$ | Sociale kost |
| $v$ | Snelheid |
| $v_f$ | Ongehinderde snelheid |
| $v_\mu$ of $\mu_v$ | Gemiddelde loopsnelheid (beide worden gebruikt) |
| $V$ | Verzameling vertices |
| $w$ | Gewicht |
| $x$ | Afstand in breedterichting |
| $y$ | Afstand in lengterichting |
| $t_{cor}$ | Gecorrigeerde looptijd pad |
| $t_{gkz}$ | Gecorrigeerde looptijd pad, aangepast naar een gekozen loopsnelheid |
| $t_{sys}$ | Looptijd pad (som van gewichten op kortste pad) |
| $\beta$ | Tijd per traptrede |
| $\gamma$ | IJkconstante |
| $\lambda$ | Tijd deur openen |
| $\sigma_v$ | Standaardafwijking loopsnelheid |

# 1 Theoretisch kader

## 1.1 Grafentheorie

In dit hoofdstuk wordt de grafentheorie geïntroduceerd, met focus op de kennis relevant aan ons onderzoek. Deze kennis is gehaald uit *Grafen: Kleuren en Routeren*, geschreven door Alexander Schrijver (z.d.).

### 1.1.1 De Graaf




*Figuur 1: voorbeeld graaf*

Een graaf is een netwerk, bestaand uit vertices (punten) en zijden (connecties tussen punten). Een netwerk zoals in figuur 1 is slechts een visuele representatie van een graaf. Eenzelfde graaf kan ook op formele wijze opgeschreven worden. Men definieert een graaf $G$ als volgt:

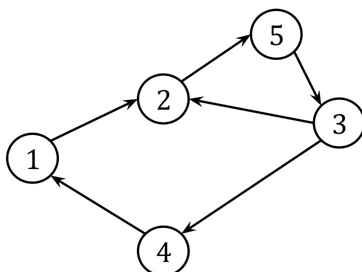
$$G := (V, E) \quad (1)$$

$$E := \{\{u,\ v\} \mid u, v \in V \; en \; u \neq v\} \quad (2)$$

Uitleg: $V$ is de verzameling vertices. $E$ is de verzameling zijden. Deze bestaat uit paren $\{u,\ v\}$, waarvoor geldt dat $u$ en $v$ beide vertices zijn uit $V$ $(u,\ v \in V)$ en $u$ en $v$ niet hetzelfde punt zijn $(u \neq v)$.

De graaf in figuur 1 kan genoteerd worden als:
$G := (\{a, b, c, d\}, \{\{a, b\}, \{a, c\}, \{b, c\}, \{b, d\}, \{c, d\}\})$.

### 1.1.2 De Gerichte Graaf




*Figuur 2: voorbeeld gerichte graaf*

Eerder was te zien dat de paren uit $E$ geen richting hadden. Een gerichte graaf zoals in figuur 2 heeft dat wel. Een zijde kan enkel één richting worden doorlopen. Een gerichte graaf $D$ wordt gedefiniëerd als volgt:

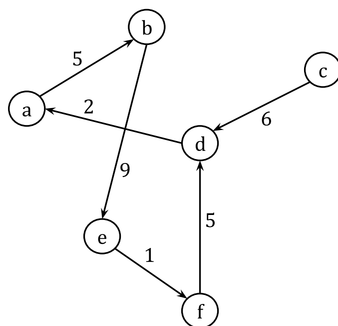
$$D := (V, A) \tag{3}$$

$$A := \{(u, v) \neq (v,\ u) \mid u, v \in V\ en\ u \neq v\} \tag{4}$$

Uitleg: $A$ is de verzameling gerichte zijden $(u,\ v)$, waarvan de zijde vertrekt uit $u$ en aankomt in $v$. $(u, v)$ en $(v, u)$ zijn dus niet hetzelfde ($(u, v) \neq (v,\ u)$).

De graaf in figuur 2 kan genoteerd worden als:
$D := (\{1, 2, 3, 4, 5\}, \{(1, 2), (2, 5), (5, 3), (3, 2), (3, 4), (4, 1)\})$

### 1.1.3 De Gewogen Graaf




*Figuur 3: voorbeeld gewogen graaf*

Bij toepassingen van een graaf worden vaak waarden gegeven aan zijden, deze heten gewichten. Men krijgt dan een gewogen graaf zoals in figuur 3. Deze kan zowel gericht als ongericht zijn. Een gewicht kan verschillende grootheden voorstellen, zoals een lengte, tijdsduur of capaciteit van een zijde. Aan elke zijde $e \in E$ wordt een gewicht $w(e)$ gegeven. Dit is een reëel getal. De functie $w$ is gedefinieerd als volgt:

$$w\colon\ E \to \mathbb{R} \tag{5}$$

Uitleg: $w$ is een functie, die een zijde $e$ uit verzameling $E$ (of $A$ als er gesproken wordt van een gerichte graaf), een reëel getal uit $\mathbb{R}$ geeft.

Het gewicht van zijde $(a, b)$ in figuur 3 wordt genoteerd als volgt: $w((a, b)) = 5$

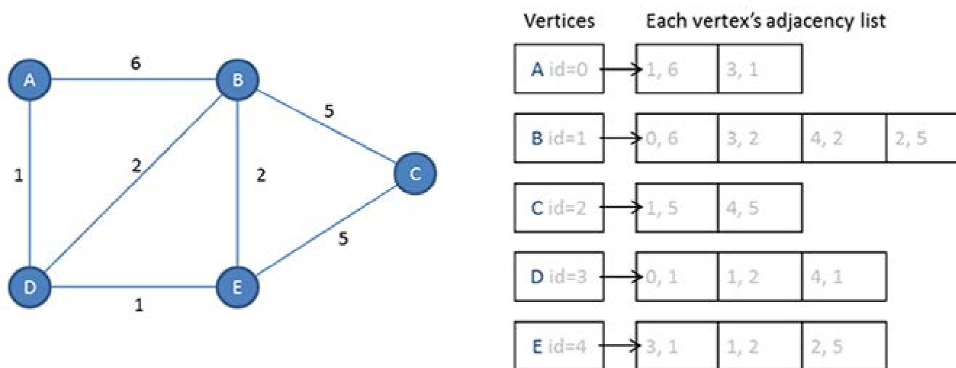
### 1.1.4 Matrixvoorstelling en adjacency list

Grafen kunnen beschreven worden in matrices. Dit noemt men een matrixvoorstelling. De rijen en kolommen geven vertices aan. In een ongewogen graaf kan een 1 genoteerd worden in een cel, om aan te geven dat er een zijde aanwezig is tussen de vertices van de bijbehorende rij en kolom. In een gewogen graaf kan het gewicht van de zijde genoteerd worden in een cel. Als er geen zijde aanwezig is tussen twee vertices, wordt er 0 genoteerd in de cel tussen de bijbehorende rij en kolom. In een gerichte graaf representeren de rijen, de vertices waaruit een zijde vertrekt. De kolommen representeren de vertices waarin de zijden aankomen.

De matrixvoorstelling van de graaf in figuur 3 wordt als volgt geschreven:

$$\begin{pmatrix} 0 & 5 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 9 & 0 \\ 0 & 0 & 0 & 6 & 0 & 0 \\ 2 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 5 & 0 & 0 \end{pmatrix} \tag{6}$$

Aan de computationele kant van de grafentheorie wordt vaak een variatie op deze matrixvoorstelling gebruikt: een adjacency list. Hieronder is een voorbeeld van een adjacency list weergeven:




*Figuur 4: voorbeeld adjacency list (Computer Science Bytes, z.d.)*

Een adjacency list noteert voor elke vertex respectievelijk de (corresponderende nummers van de) verbonden vertices en de gewichten van de verbindende zijden in geordende lijsten. In het voorbeeld is $C$ (met nummer 2) verbonden met $B$ (met nummer 1) en is het gewicht $w(\{C, B\}) = 5$. Er wordt dus $(1, 5)$ naast nummer 2 genoteerd om aan te geven de zijde tussen $B$ en $C$ aan te geven.

In Python wordt de adjacency list uit figuur 4 als volgt genoteerd:

```
graph = {
    0 : [[1,6],[3,1]],
    1 : [[0,6],[3,2],[4,2],[2,5]],
    2 : [[1,5],[4,5]],
    3 : [[0,1],[1,2],[4,1]],
    4 : [[3,1],[1,2],[2,5]]
}
```

De reden voor het gebruik van een adjacency list in plaats van een matrixvoorstelling is het feit dat een relatief grote graaf in matrixvoorstelling, in veel gevallen een ijle (grotendeels lege) matrix oplevert. De code van een algoritme (zoals Dijkstra's algoritme) zal dan veel lege cellen moeten afgaan. Een adjacency list geeft alleen de bestaande zijden weer en is dus sneller.

## 1.2 Dijkstra’s algoritme

Het kortste pad algoritme van Edsger Dijkstra (1959) vindt op systematische wijze het kortste pad[1] van een bepaald startpunt naar een bepaald eindpunt in een gewogen graaf, mits de gewichten positief zijn. Het werkt op zowel gerichte als ongerichte grafen. Dijkstra’s algoritme werkt als volgt:

Neem een graaf $G := (V, E)$ en neem een startpunt en een eindpunt op deze graaf. Deze worden respectievelijk $s$ en $t$ genoemd.

Men start met elke vertex op deze graaf $v \in V$ markeren als "onbezocht". Elke vertex krijgt een *afstand tot het startpunt*, dit is de som van de gewichten van het pad naar het startpunt. Deze is nul voor het startpunt en (voor nu) oneindig voor de andere vertices.

Daarnaast krijgt elke vertex een *vorige vertex*: neem een pad $(a,\ b,\ c,\ d)$, dan heeft vertex $d$ *vorige vertex* $c$, vertex $c$ *vorige vertex* $b$, etc. Het noteren van deze *vorige vertices* is essentieel om achteraf het snelste pad te reconstrueren. Aan het begin zijn nog geen *vorige vertices* toegewezen, aangezien er nog geen paden bepaald zijn.

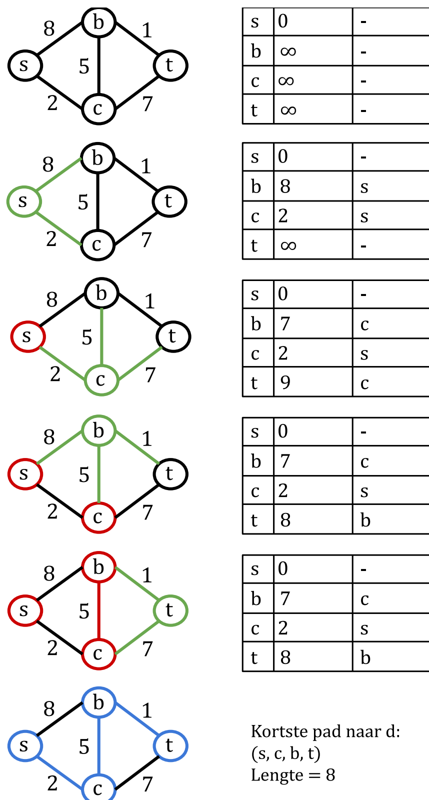
Men begint met het bezoeken van startpunt $s$. Dit houdt in dat alle verbonden vertices $a \mid \{s,\ a\} \in E$ worden gecheckt: De *afstand tot het startpunt* voor alle $a$ wordt veranderd naar het gewicht van de zijde die hen verbindt $w(\{s, a\})$. Daarnaast wordt $s$ genoteerd als de *vorige vertex* voor de verbonden vertices $a$. Het startpunt kan nu gemarkeerd worden als bezocht.

Vervolgens neemt men de onbezochte vertex met de kleinste *afstand tot het startpunt*. Deze vertex $u$ wordt nu bezocht: Vanuit $u$ wordt elke verbonden vertex $v \mid \{u,\ v\} \in E$ gecheckt: Neem de som van de *afstand tot het startpunt* van vertex $u$ en het gewicht van de verbindende zijde $w(\{u, v\})$. Is deze kleiner dan de, tot dusver bepaalde, *afstand tot het startpunt* van $v$? Als dit zo is, dan heeft het algoritme een korter pad gevonden naar $v$. De *afstand tot het startpunt* van vertex $v$ verandert dus naar deze som. Ook wordt $u$ als vorige vertex van $v$ genoteerd. Is dit niet het geval, dan verandert er niets. Er is dan immers geen korter pad gevonden. Als elke verbonden vertex $v$ gecheckt is, wordt vertex $u$ als bezocht gemarkeerd.

Opnieuw wordt de onbezochte vertex met de kleinste *afstand tot het startpunt* genomen. Van deze vertex $u$ worden de verbonden vertices $v$ gecheckt. Ten slotte wordt vertex $u$ als bezocht gemarkeerd. Dit proces wordt herhaald tot het eindpunt $t$ bezocht is.

De afstand tot $s$ van $t$ is nu minimaal, dus het kortste pad is gevonden. Dit kortste pad kan gereconstrueerd worden door van $t$ de *vorige vertex* te nemen, hiervan de *vorige vertex* te nemen, etc. totdat men bij $s$ uitkomt. Deze reeks aan *vorige vertices* vormt het kortste pad tussen het startpunt $s$ en het eindpunt $t$.

[1] De formele definitie van een pad volgt in de volgende paragraaf, deze is voor nu nog niet nodig.



| s | 0 | - |
|---|---|---|
| b | ∞ | - |
| c | ∞ | - |
| t | ∞ | - |

| s | 0 | - |
|---|---|---|
| b | 8 | s |
| c | 2 | s |
| t | ∞ | - |

| s | 0 | - |
|---|---|---|
| b | 7 | c |
| c | 2 | s |
| t | 9 | c |

| s | 0 | - |
|---|---|---|
| b | 7 | c |
| c | 2 | s |
| t | 8 | b |

| s | 0 | - |
|---|---|---|
| b | 7 | c |
| c | 2 | s |
| t | 8 | b |

Kortste pad naar d:
(s, c, b, t)
Lengte = 8

*Figuur 5: demonstratie Dijkstra's algoritme*

In de bovenstaande figuur wordt een demonstratie gegeven van Dijkstra's kortste pad algoritme. Aan de rechterzijde is een tabel die voor elke vertex (1e kolom) de afstand tot het startpunt weergeeft (2e kolom) en de vorige vertex in het pad naar het startpunt (3e kolom) In eerste instantie wordt vertex $s$ bezocht en de verbonden vertices $b$ en $c$ gecheckt (de verbindende zijden zijn groen). De tabel aan de rechterzijde weergeeft de gegevens die uit deze stap volgen. Vervolgens wordt $s$ als bezocht gemarkeerd (deze wordt rood). Vervolgens wordt uit de gegevens van de tabel een onbezochte vertex gekozen met de kortste afstand tot het startpunt, dit is vertex $c$ en deze wordt nu bezocht. Dit wordt herhaald totdat ook eindpunt $t$ bezocht is. Vervolgens wordt uit de rechterkolom een route geconstrueerd: vertex $t$ heeft vorige vertex $b$, die heeft weer vorige vertex $c$ en die heeft vorige vertex $s$. Hieruit volgt dus het pad $(s, c, b, t)$ met lengte 8, want de *afstand tot het startpunt* van $t$ is 8.

Opmerkingen:

- Indien de graaf gericht is, kan vanuit een vertex $u$, alleen vertices worden gecheckt die verbonden zijn door een zijde die vertrekt uit $u$.
- Een eindpunt kan ook gekozen worden zodra elke vertex bezocht is. In dit geval is iedere afstand tot het startpunt minimaal. Deze methode duurt echter langer.

## 1.3 Stromen

In november 2024 hebben wij meegedaan aan twee masterclasses van *NETWORKS goes to school*, een deel van het *NETWORKS* onderzoeksprogramma. Deze gingen over de formele kant van verkeerstheorie, met toepassingen van speltheorie en grafentheorie. In dit hoofdstuk worden de (voor dit onderzoek) relevante elementen uit deze masterclasses samengevat. De bron voor dit hoofdstuk is het boekje *NETWORKS goes to school #3* (2024).

### 1.3.1 s,t-stroom

Een wandeling $W$ van een vertex $s$ naar vertex $t$ op een graaf (formule 1), is een vector bestaand uit vertices, zodanig dat er een zijde is die achtereenvolgende vertices verbindt. Het eerste element uit deze vector is vertex $s$ en de laatste vertex $t$. Een wandeling wordt als volgt gedefinieerd:

$$W := (v_0, v_1, \dots, v_r) \mid (\{v_{q-1}, v_q\} \in E \mid q = 1, \dots, r), v_0 = s, v_r = t \quad (7)$$

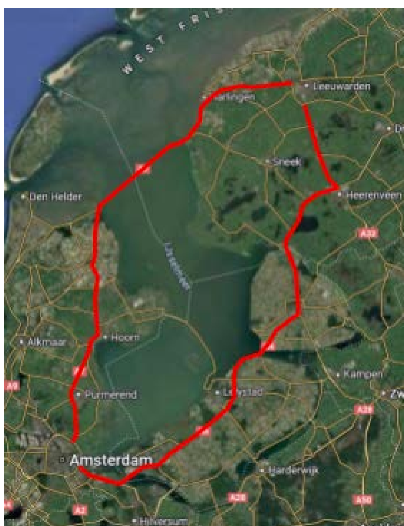
Een pad $P$ is een wandeling, zodanig dat geen enkele vertex twee keer voorkomt in de vector:

$$P := (v_0, v_1, \dots, v_r) \mid (\{v_{q-1}, v_q\} \in E \wedge \forall q \neq p\ (v_q \neq v_p) \mid q, p = 1, \dots, r), v_0 = s, v_r = t \quad (8)$$

Het is mogelijk dat er tussen twee vertices $s$ en $t$ geen wandeling en dus geen pad bestaat. Men neemt dus een vertex $s$ en vertex $t$ waartussen minimaal één pad is. We definiëren $\mathcal{P}$ als de verzameling van alle paden tussen $s$ en $t$:

$$\mathcal{P} := \{P_1, \dots, P_k\}, \ met\ k > 0 \quad (9)$$

Over deze paden gaat een stroom. Beschouw bijvoorbeeld een stroom als een stroom van auto’s die van Amsterdam naar Leeuwarden gaan. Twee mogelijke paden op dit traject zouden via de A6 of de afsluitdijk gaan:




*Figuur 6: mogelijke paden tussen Amsterdam en Leeuwarden (Google maps, 2025)*

Over elk $P_i \in \mathcal{P}$ gaat een deelstroom $f_{Pi}$, opdat de som van deelstromen gelijk is aan één.

$$\sum_{i=1}^{k} f_{Pi} = 1 \tag{10}$$

Een deelstroom heeft dus een waarde tussen en inclusief 0 en 1.

$$f_{Pi}: P_i \rightarrow [0, 1] \tag{11}$$

De (totale) stroom tussen vertex $s$ en vertex $t$ is een vector $f$:

$$f := (f_{P1}, f_{P2}, \dots, f_{Pk}) \tag{12}$$

De reden dat er gekozen wordt voor een representatie van een continue stroom (reële getallen tussen 0 en 1), in tegenstelling tot een discrete benadering (gehele getallen) is het feit dat het aantal voertuigen binnen een stroom zeer groot is en de impact van een individuele deelnemer verwaarloosbaar is.

Ten slotte wordt de verzameling mogelijke stromen $F$ als volgt gedefinieerd:

$$F := \{f \in \mathbb{R}^k \mid f_{Pi} \geq 0, \sum_{i=1}^{k} f_{Pi} = 1\} \tag{13}$$

In het geval van een auto-stroom tussen Amsterdam en Leeuwarden (zie figuur 6) is de reistijd op een wegstuk, de vertraging[2], afhankelijk van de stroomgrootte over dit wegstuk. Als iedereen bijvoorbeeld besluit over de Afsluitdijk te gaan, ontstaan er files en is er hogere vertraging. Dan is op de A6 een lagere vertraging.

Voor elke zijde van de graaf wordt dus een vertragingsfunctie $L_e \mid e \in E$ toegewezen. Een vertraging is een functie van de stroom over de bijbehorende zijde.

$$L_e : f_e \rightarrow \mathbb{R}^+ \tag{14}$$

De totale vertraging op een pad $i$ is dan:

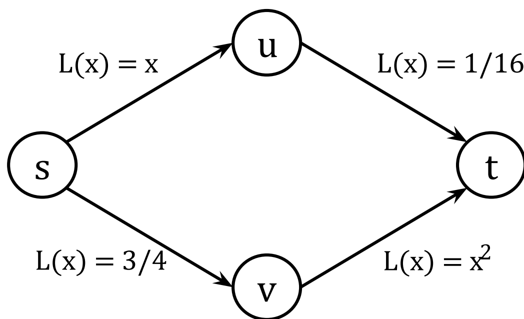
$$L_{Pi}(f_{Pi}) = \sum_{e \in Pi} L_e(f_e) \tag{15}$$

De sociale kost $SC$, of totale vertraging, van een $s, t$ -stroom is vergelijkbaar met de verwachtingswaarde in een kansexperiment en wordt gegeven door:

$$SC(f) = \sum_{i=1}^{k} L_{Pi}(f_{Pi}) f_{Pi} \tag{16}$$

[2] Met vertraging wordt niet de extra tijd (door file) om van één punt naar de andere te gaan bedoeld, maar de totale tijdsduur.

Voorbeeld:




*Figuur 7: graaf met vertragingsfuncties*

In figuur 7 staat een graaf met een $s, t$ -stroom en vertragingsfuncties, de $x$ daarin betekent de stroom over die zijde. De twee paden van $s$ naar $t$ zijn: $P_1 = (s, u, t)$ en $P_2 = (s, v, t)$. Een mogelijke stroom is $f = (\frac{1}{2}, \frac{1}{2})$. De vertraging op pad 1 is in dit geval $L_{P1}(\frac{1}{2}) = \sum_{e \in P1} L_e(f_e) = \frac{1}{2} + \frac{1}{16} = \frac{9}{16}$. De vertraging op pad 2 is $L_{P2}(\frac{1}{2}) = \sum_{e \in P2} L_e(f_e) = \frac{3}{4} + (\frac{1}{2})^2 = 1$. De sociale kost van deze stroom is $SC(\frac{1}{2}, \frac{1}{2}) = L_{P1}(f_{P1})f_{P1} + L_{P2}(f_{P2})f_{P2} = \frac{9}{16} \cdot \frac{1}{2} + 1 \cdot \frac{1}{2} = \frac{14}{19}$.

## 1.3.2 Nash-stroom en optimale stroom

Een Nash-stroom haalt zijn oorsprong uit het Nash-evenwicht, een Nash-evenwicht is “een uitkomst in een niet-coöperatief spel voor twee of meer spelers waarbij geen enkele speler zijn verwachte uitkomst kan verbeteren door alleen zijn eigen strategie te veranderen.” (Eldridge, z.d.). In een Nash-stroom kan niemand een sneller pad volgen. Voor een Nash-stroom $f_{nash} \in F$ geldt dus:

$$voor\ \ f_{P_i}, f_{P_j} \in f_{nash}\ \ geldt: \begin{cases} f_{P_i}, f_{P_j} > 0: & L_{P_i}(f_{P_i}) = L_{P_j}(f_{P_j}) \\ f_{P_i} > 0\ \wedge\ f_{P_j} = 0: & L_{P_i}(f_{P_i}) \le L_{P_j}(f_{P_j}) \end{cases} \tag{17}$$

Dit houdt in dat voor twee deelstromen in een Nash-stroom twee opties bestaan:

- Als over beide paden een deelstroom groter dan nul gaat, zal de totale vertraging op deze zijden gelijk moeten zijn. Als deze niet gelijk zijn, is er voor een deel van het verkeer een sneller pad beschikbaar waar zij beter overheen kunnen gaan, wat in tegenspraak is met het idee een Nash-stroom.
- Indien er over een pad geen deelstroom gaat, moet de vertraging op dat pad minstens zo groot zijn als de vertraging op elk pad waar wel verkeer overheen gaat. Dit betekent dat een pad met een deelstroom van 0 een arbitrair hoge vertraging mag hebben, zonder dat dit in strijd is met het evenwicht. Denk bijvoorbeeld aan de verbinding tussen Amsterdam en Leeuwarden: ook al is er file op de Afsluitdijk en de

A6, een pad via Eindhoven is vrijwel altijd langzamer. Dit geeft aan dat de vertraging op ongebruikte paden niet noodzakelijk gelijk hoeft te zijn aan die op de gebruikte paden, maar alleen minimaal even groot of hoger.

Aan een situatie waarin twee deelstromen gelijk zijn aan 0 hoeven geen eisen te worden gesteld omdat er toch geen verkeer is dat van strategie (pad) kan wisselen.

In de ogen van een sociaal planner is het bevorderlijk om de sociale kost minimaal te houden. Hiervoor wordt de optimale stroom $f^*$ geïntroduceerd. Voor de optimale stroom geldt dat de sociale kost minimaal is:

$$f^* := arg\ min_{f \in F}\ SC(f) \tag{18}$$

De prijs van anarchie (kortweg $PoA$) tussen vertices $s$ en $t$ is de verhouding van de sociale kost van de Nash-stroom over de sociale kost van de optimale stroom. Deze wordt gegeven door:

$$PoA\ = \frac{SC(f_{nash})}{SC(f^*)} \tag{19}$$

Hoe lager de prijs van anarchie, hoe beter een netwerk is ingericht. Het minimum van de prijs van anarchie is één, deze zal nooit lager worden. Dit volgt uit het feit dat de sociale kost van de Nash-stroom in het beste geval gelijk zal zijn aan die van de optimale stroom, maar nooit lager. De optimale stroom is immers optimaal. In het beste geval zullen dus de teller en noemer gelijk zijn dus de prijs van anarchie één.

Voorbeeld:

Voor de Nash-stroom in figuur 7 $f_{nash} = (f_{P1}, f_{P2})$ geldt dus $L_{P1}(f_{P1}) = L_{P2}(f_{P2})$.[3] Omdat er maar twee mogelijke stromen zijn geldt dat als $f_{P1} = x$ dan $f_{P2} = 1 - x$. Door dit in te vullen in de vertragingsfuncties in figuur 7 kan de volgende vergelijking worden verkregen: $x + \frac{1}{16} = \frac{3}{4} + (1 - x)^2$. Meegenomen dat $0 \le x \le 1$ kan de oplossing $x = \frac{3}{4}$ gevonden worden. De Nash-stroom van figuur 7 is dus $f_{nash} = (\frac{3}{4}, \frac{1}{4})$.

Voor de optimale stroom in deze figuur $f^* = (f_{P1}, f_{P2})$, neemt men weer $f_{P1} = x^*$ en $f_{P2} = 1 - x^*$. Dit geeft $SC(x^*, 1 - x^*) = x^*(x^* + \frac{1}{16}) + (\frac{3}{4} + (1 - x^*)^2)(1 - x^*)$. De sociale kost moet minimaal zijn dus $\frac{dSC}{dx^*} = -3(x^*)^2 + 8x^* - 3\frac{11}{16} = 0$. Hieruit volgt dat $x^* \approx 0,59$. Dit geeft $\frac{d^2SC}{dx^{*2}} > 0$ dus dit is daadwerkelijk een minimum. Dus de optimale stroom is $f^* = (0,592...\ ; 0,407...)$.

$f^* = (0,592...\ ; 0,407...)$ geeft $SC(f^*) = 0,761...$ en $f_{nash} = (\frac{3}{4}, \frac{1}{4})$ geeft $SC(f_{nash}) = \frac{15}{16}$.

Hieruit volgt dat de prijs van anarchie van de graaf in figuur 7 $PoA = \frac{\frac{15}{16}}{0,761...} \approx 1,23$ is.

[3] Er wordt uitgegaan van de eerste optie van een Nash-stroom. Als dit onterecht zou zijn, zou de vergelijking geen kloppende oplossing opleveren.

## 1.4 Verkeerstheorie

Zoals in de vorige paragraaf beschreven, is er een relatie tussen vertraging en drukte op een zijde. In de voorbeelden werden zeer versimpelde vertragingsfuncties gebruikt, zoals $L(x) = x$ of $L(x) = \frac{3}{4}$. Echter moeten er realistische formules worden gebruikt bij toepassing van vertragingsfuncties in de werkelijkheid. Hiervoor is het essentieel om te verdiepen in verkeerstheorie.

### 1.4.1 Fundamenteel verband

Beschouw een weg als een eendimensionale lijn met voertuigen erop die één richting op bewegen. Neem aan dat deze voertuigen met gemiddelde snelheid $v$ (in $m/s$) bewegen over de lengte $y$ (in $m$) van deze lijn in de tijd $t$ (in $s$):

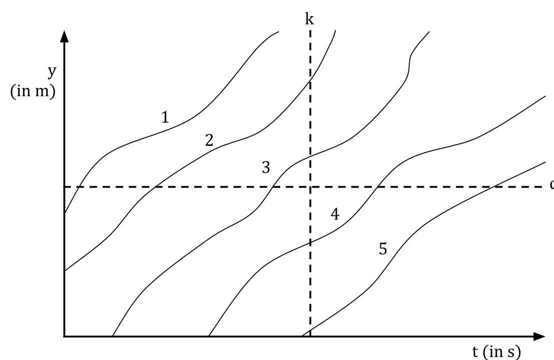
$$v = \frac{y}{t} \quad (20)$$

Over deze lengte van het lijnstuk zijn $n$ voertuigen. De eendimensionale dichtheid $k$ van deze voertuigen is het aantal voertuigen ($veh$) per meter op één moment in de tijd:

$$k = \frac{n}{y} \quad (21)$$

Stel je nu voor dat je langs een weg staat. De intensiteit (of stroomsnelheid) $q$ (in $veh/s$) van een stroom is het aantal voertuigen dat jou passeert per tijdseenheid op één plek:

$$q = \frac{n}{t} \quad (22)$$




*Figuur 8: (afstand, tijd) - diagram van een verkeersstroom*

In de bovenstaande figuur staat de beweging van een verkeersstroom weergeven in een (afstand, tijd) - diagram. Elke lijn is een voertuig dat in de tijd $t$ beweegt over de lengte $y$ van de weg. De snelheid van een voertuig is de helling van die lijn. De dichtheid is het aantal voertuigen op één moment in de tijd, over de lengte. De Intensiteit is het aantal voertuigen dat langskomt op één plek in de lengte, over de tijd.

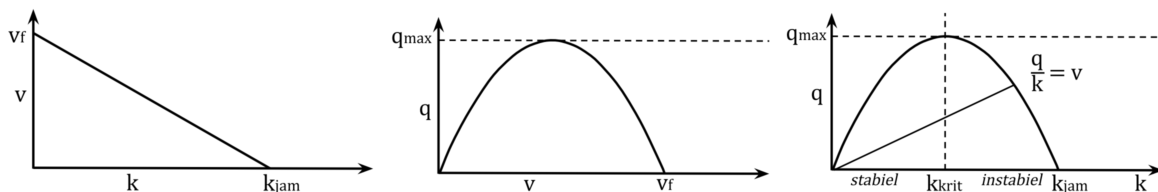
Tussen de snelheid, de intensiteit en de dichtheid bestaat het fundamentele verband, deze valt af te leiden uit formules 20, 21 en 22:

$$q = v \cdot k \quad (23)$$

(Hoogendoorn, 2016)

## 1.4.2 Fundamentele diagrammen

De relatie tussen elke twee van de grootheden snelheid, intensiteit en dichtheid wordt beschreven aan de hand van zogeheten fundamentele diagrammen (figuur 9), geïntroduceerd door Greenshields (1935). Deze diagrammen weergeven de karakteristieken van verkeersstromen en zijn geen universele waarheden. De relatie wordt hierin beschreven op macroscopisch niveau. Goed om te onthouden is dat uit elk van de drie diagrammen de andere twee kunnen worden afgeleid.




*Figuur 9: Greenshields fundamentele diagrammen*

Om deze diagrammen beter te begrijpen, is het handig om naar een typische dag op de snelweg te kijken:

- Lage dichtheid: Stel je een snelweg rond middernacht voor. Er is weinig verkeer op de weg, waardoor de dichtheid $k$ laag is. Bestuurders kunnen met ongehinderde snelheid $v_f$ rijden. Maar omdat er weinig verkeer is, is de intensiteit $q$ ook laag. Men spreekt hier van een stabiele verkeersstroom.
- Toenemende dichtheid: In de ochtend neemt het verkeer toe. De eerste mensen gaan naar hun werk, waardoor de dichtheid stijgt. De snelheid daalt lichtelijk omdat auto's iets dichter op elkaar rijden. De intensiteit stijgt. Op een bepaald moment bereikt de intensiteit zijn maximum $q_{max}$, wat overeenkomt met een kritieke dichtheid $k_{krit}$. Dit is het punt waarop de weg het meeste verkeer per tijdseenheid verwerkt.
- Overbelasting: Als er nog meer auto's de weg op komen, wordt het steeds lastiger om soepel door te rijden. Bestuurders moeten vaker remmen en optrekken, waardoor de snelheid sneller afneemt dan de dichtheid toeneemt. Dit leidt ertoe dat de intensiteit begint te dalen, ondanks dat er veel auto's op de weg zijn. Men spreekt van een instabiele verkeersstroom, waarin kleine verstoringen (zoals een auto die van rijstrook wisselt) kunnen leiden tot filevorming.
- File: Als de dichtheid blijft stijgen, bereikt deze een maximum $k_{jam}$. Het verkeer staat dan stil, bijvoorbeeld tijdens de spits. De snelheid is $v = 0$, dus de intensiteit is ook $q = 0$, omdat er geen auto's meer doorstromen. Er is dan sprake van file.

Voor het linker diagram (en daarmee alle drie) zijn veel verschillende modellen voorgesteld. Greenshields (1935) stelde zelf het volgende simpele model voor:

$$v = v_f(1 - \frac{k}{k_{jam}}) \tag{24}$$

### 1.4.3 Voetgangersstromen

Toch blijkt de breedte van de weg soms ook van belang te zijn. Er kan zonder fundamentele veranderingen een extra dimensie toegevoegd worden: de weg krijgt breedte $x$ (in $m$). De intensiteit $q'$ wordt nu dus gegeven in voertuigen per tijdseenheid per breedtemeter:

$$q' = \frac{n}{t \cdot x} \tag{25}$$

De dichtheid $k'$ wordt gegeven door voertuigen per vierkante meter:

$$k' = \frac{n}{x \cdot y} \tag{26}$$

De snelheid blijft hetzelfde (formule 20). Ook geldt eenzelfde fundamenteel verband:

$$q' = v \cdot k' \tag{27}$$

Het fundamentele diagram is hetzelfde voor het een- en tweedimensionale verband (aangezien er bij het tweedimensionale verband simpelweg een constante $x$ is toegevoegd).

In zijn proefschrift stelde Kladek (1966) het volgende model voor, inzake de relatie tussen dichtheid $k'$ (in $veh/m^2$) en snelheid $v$ (in $m/s$) binnen een verkeersstroom:

$$v = v_f \left(1 - e^{-\gamma\left(\frac{1}{k'} - \frac{1}{k'_{jam}}\right)}\right) \tag{28}$$

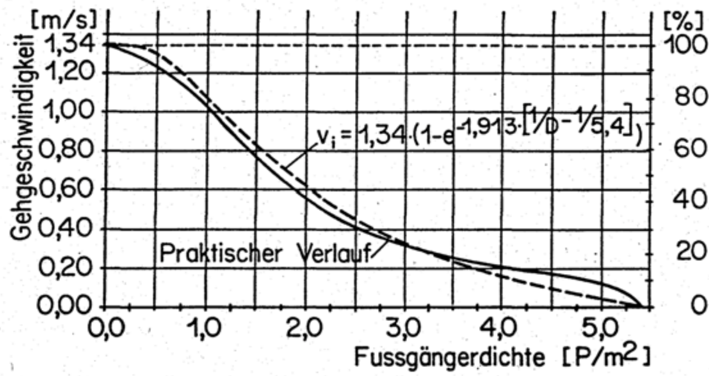
De ijkconstante $\gamma$ in (in $veh/m^2$), de opstoppingsdichtheid $k'_{jam}$ (in $veh/m^2$) en de vrije snelheid $v_{free}$ (in $m/s$) zijn constanten, uniek aan het soort stroom (auto’s, voetgangers, etc.).

Later, in een onderzoek naar transporttechniek van voetgangers geeft Weidmann (1993) waarden aan deze constanten voor voetgangersstromen. $P$ betekent personen.

$$\gamma = 1,913\ P/m^2 \tag{29}$$

$$k'_{jam} = 5,4\ P/m^2 \tag{30}$$

$$v_f = 1,34\ m/s \tag{31}$$




*Figuur 10: verband snelheid en voetgangersdichtheid, volgens de functie van Kladek en constanten van Weidmann. (Weidmann, 1993)*

Door de constanten en formules uit deze paragraaf samen te voegen kan een realistische vertragingsfunctie worden afgeleid. Deze formule kan gebruikt worden om vertragingsfuncties te plaatsen op de zijden van de graaf van het schoolgebouw. De afleiding van deze formule en het gebruik hiervan volgt in paragraaf 4.1.

# 2 Graaf Hyperion Lyceum

Met plattegronden van het Hyperion Lyceum kan een graaf geconstrueerd worden. In feite wordt er een model gemaakt van looproutes en verbindingen in het schoolgebouw van het Hyperion Lyceum. Eerst is de structuur bepaald en een ongewogen graaf gemaakt. Vervolgens zijn voor de zijden gewichten berekend. De ongewogen graaf en gewogen graaf zijn te vinden in bijlage B en bijlage C, respectievelijk.

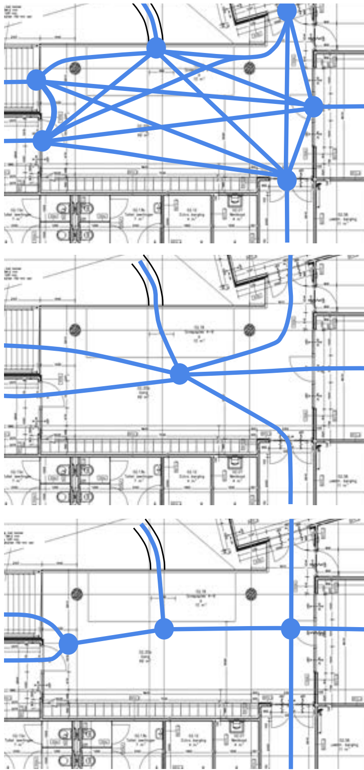
## 2.1 Verantwoording

### 2.1.1 Relevante locaties

De doelgroep voor het navigatiesysteem bestaat uit leerlingen en bezoekers van het Hyperion Lyceum. De selectie van begin- en eindpunten moet hierop afgestemd zijn. Hiermee zijn plekken waar leerlingen niet mogen komen uitgesloten. Het is bijvoorbeeld niet mogelijk om door de lerarenkamer te lopen (wel is er een vertex vóór de lerarenkamer). Alle andere locaties waar een leerling mogelijk naar toe zou moeten zijn wel beschikbaar. Hieronder vallen bijvoorbeeld klaslokalen en kantoren (van de decaan, de roostermaker, etc.). Ook zijn er vertices toegevoegd voor kruispunten en splitsingen, dit is essentieel voor de structuur van de graaf.

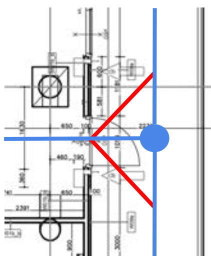
### 2.1.2 Simpliciteit versus granulariteit

Het omzetten van een plattegrond naar een graaf vereist abstrahering. Er moet een balans gevonden worden tussen granulariteit (precisie) en simpliciteit. Aan de ene kant is het belangrijk dat de graaf waarheidsgetrouw is. Het systeem kan dan een goede schatting maken van het snelste pad en de looptijd hiervan. Hier komt bij kijken dat meer waarheidsgetrouwheid om een hogere complexiteit vraagt. Aan de andere kant moet de simpliciteit van de graaf gewaarborgd worden. De graaf moet handmatig getekend en ingevoerd worden. Daarnaast moeten alle zijden gemeten en berekend worden. Een overvloed aan vertices en zijden zal deze processen enorm vertragen.


*Figuur 11: drie verschillende maten van simplificatie in een hal*

In figuur 11 is de plattegrond van een hal op de tweede verdieping te zien. Deze heeft 6 (relevante) aansluitingen: Drie lokalen, een gang, een trap en de glijbaan. Bovenaan is een compleet waarheidsgetrouwe graaf te zien waarbij elke kortste route tussen twee aansluitingen mogelijk is gemaakt met een zijde, resulterend in een groot aantal vertices en zijden. Daaronder is de hal versimpeld tot één vertex met één zijde voor elke aansluiting. Op een route die over de trap, het lokaal direct ernaast in gaat, zou het systeem de looptijd overschatten. Onderaan is een voorgestelde structuur weergeven met een balans tussen simpliciteit en granulariteit

In figuur 11 is een enkel geval van het dilemma tussen simpliciteit en waarheidsgetrouwheid weergeven. In werkelijkheid zijn alle vertices in zekere mate *dilemmaplekken*. Elke vertex heeft namelijk kortere routes, zo kunnen bijvoorbeeld de hoeken worden afgesneden op de voordehandliggende T-splitsing in figuur 12. De graaf bestaat uit ongeveer 160 vertices, dus zijn er ook 160 dilemmaplekken. Deze dilemmaplekken zorgen ieder voor overschattingen van de looptijd van een route, en zullen dus verrekend moeten worden in de tijd die het systeem aangeeft.


*Figuur 12: dilemmaplek*

Er wordt voorgesteld om de graaf te tekenen in een logische structuur. Dit houdt in dat elke relevante locatie een vertex krijgt toegewezen op het gebied van deze locatie. Daarnaast wordt elk kruispunt en elke dergelijke splitsing gerepresenteerd als een vertex. Twee

vertices kunnen alleen worden samengevoegd als deze minder dan twee meter uit elkaar liggen. Zo kan een waarheidsgetrouwe structuur behouden worden, zonder de graaf extreem gecompliceerd te maken.

Tevens moet het systeem gecorrigeerd worden voor het omlopen op deze dilemmaplekken: Tijd $dt_{dp}$ (in $s$) is het verschil in tijd tussen de waarheidsgetrouwe route en de versimpelde route (volgens de zijden van de graaf) op één dilemmaplek, dus de overschatting van het systeem op één dilemmaplek. $dt_{gem}$ is het gemiddelde van deze verschillen. Om de aangegeven looptijd $t_{sys}$ (berekend met Dijkstra's algoritme) te corrigeren moet dit gemiddelde worden afgetrokken van deze tijd, voor het aantal vertices $n_v$ op het pad (min twee want het startpunt en eindpunt zijn ook vertices op het pad maar geen dilemmaplekken want ze kunnen niet afgesneden worden). Hieruit ontstaat de gecorrigeerde tijd $t_{cor}$.

$$dt_{gem} = \frac{\Sigma\, dt_{dp}}{n} \quad (32)$$

$$t_{cor} = t_{sys} - dt_{gem} \cdot (n_v - 2) \quad (33)$$

Door metingen te doen op twaalf willekeurig gekozen vertices op de graaf van de school (bijlage B) kan de volgende data gevonden worden. De waarheidsgetrouwe looptijd is gemeten door maximaal af te snijden ten opzichte van de aangegeven graafstructuur.

| Dilemmaplek[4] | Waarheidsgetrouwe looptijd (in $s$) | Aangegeven looptijd (in $s$) | $dt_{dp}$ (in $s$) |
|---|---|---|---|
| 2, 3, 14 | 4,20 | 5,47 | 1,27 |
| 0, 1, 18 | 1,26 | 1,68 | 0,42 |
| 22, 24, 23 | 4,21 | 6,09 | 1,88 |
| 11, 12, 13 | 5,26 | 6,52 | 1,26 |
| 62, 63, 66 | 2,73 | 4,21 | 1,48 |
| 43, 45, 46 | 9,04 | 9,46 | 0,42 |
| 75, 76, 114 | 2,52 | 2,73 | 0,21 |
| 88, 89, 90 | 9,25 | 9,25 | 0,00 |
| 93, 94, 95 | 9,67 | 11,56 | 1,89 |
| 108, 109, 105 | 2,31 | 3,15 | 0,84 |
| 142, 140, 141 | 2,52 | 3,36 | 0,84 |
| 137, 136, 135 | 4,42 | 5,27 | 0,85 |

*Tabel 1: metingen dilemmaplekken*

[4] Geschreven als: verbonden vertex, vertex waar de dilemmaplek ligt, verbonden vertex

Uit de metingen en formule 32 volgt dat:

$$dt_{gem} = 0,95\ s \tag{34}$$

*Opmerking: Voor het vinden van een balans binnen het dilemma tussen simpliciteit en granulariteit is een alternatieve zelfbedachte methodiek uitgewerkt. Deze is te vinden in bijlage A. Echter is besloten om dit niet te gebruiken omdat het inconsistenties oplevert.*

### 2.1.3 De glijbaan

Een kenmerkend aspect van het Hyperion Lyceum is de glijbaan, gelegen te midden van het gebouw (zie figuur 13). Dit vormt een noemenswaardig aspect in de graaf. Elke gang, trap en deur kan namelijk in beide richtingen doorlopen worden, in tegenstelling tot deze glijbaan. De glijbaan kan in slechts één richting afgegleden worden en vormt dus de enige gerichte zijde in de gehele graaf. Dit vormt geen probleem aangezien de graaf uiteindelijk in een adjacency list wordt weergegeven. Daarnaast werkt Dijkstra's algoritme ook op gerichte grafen.

*Figuur 13: glijbaan Hyperion Lyceum*

## 2.2 De graaf

Door de genoemde ideeën toe te passen is een (nog ongewogen) graaf ontworpen. Een visualisering hiervan is te vinden in bijlage B.

## 2.3 Gewichten

De gewichten van de graaf worden bepaald door deze op de plattegrond te meten en om te rekenen naar tijdseenheden. De gewichten zijn dus simpelweg de looptijden om van één vertex naar de ander te gaan. In dit proces moeten er twee constanten worden bepaald: de extra tijd om een deur te openen en de looptijd per traptrede. Daarnaast moeten individueel de zijde van de glijbaan (men noteert deze zijde als $(85,\ 7)$) en de zijde van de ingang van de fietsenkelder $\{14,\ 3\}$ gemeten worden. Deze laatste is namelijk een significant minder steile trap dan de andere trappen.

### 2.3.1 Deuren openen

Allereerst de metingen van het openen van een deur. Deze wordt gevonden door de tijd te meten van een traject waarin een deur geopend moet worden. Tevens wordt de tijd gemeten van hetzelfde traject waarin de deur al open staat. Het verschil hiertussen is de tijd $\lambda$ (in $s$) om de deur te openen.

$$\lambda = t_{met\ deur} - t_{zonder\ deur} \tag{35}$$

Er is gemeten bij twee verschillende deuren:

| | Deur 1 | | | Deur 2 | | |
|---|---|---|---|---|---|---|
| | $t_{met\ deur}$ (in $s$) | $t_{zonder\ deur}$ (in $s$) | λ (in $s$) | $t_{met\ deur}$ (in $s$) | $t_{zonder\ deur}$ (in $s$) | λ (in $s$) |
| duwen (Dylan) | 12,86 | 12,45 | 0,41 | 6,90 | 5,97 | 0,93 |
| trekken (Dylan) | 13,29 | 12,31 | 0,98 | 7,33 | 6,17 | 1,16 |
| duwen (Tobias) | 14,48 | 13,46 | 1,02 | 7,28 | 6,20 | 1,08 |
| trekken (Tobias) | 15,04 | 13,01 | 2,03 | 7,21 | 6,39 | 0,82 |

*Tabel 2: metingen deuren openen*

Door het gemiddelde te nemen van de gemeten verschillen, wordt de tijdsconstante λ voor het openen van een deur geschat op 1,05 seconden. Het verschil tussen een deur open trekken en open duwen is niet in acht genomen aangezien deze nogal klein is.

$$\lambda = 1,05\ s \qquad (36)$$

## 2.3.2 Trappen lopen

De looptijd per traptrede β (in $s/trede$) kan bepaald worden door de looptijd $t_{trap}$ (in $s$) op een trap te meten en deze te delen door het aantal treden $n_{treden}$:

$$\beta = \frac{t_{trap}}{n_{treden}} \qquad (37)$$

Er zijn metingen gedaan op twee verschillende trappen:

| | Trap 1 | Trap 2 |
|---|---|---|
| $t_{trap}$ naar boven (Dylan) (in $s$) | 12,97 | 10,14 |
| $t_{trap}$ naar boven (Dylan) (in $s$) | 13,07 | 10,78 |
| $t_{trap}$ naar beneden (Dylan) (in $s$) | 12,14 | 9,82 |
| $t_{trap}$ naar beneden (Dylan) (in $s$) | 13,12 | 10,07 |
| $t_{trap}$ naar boven (Tobias) (in $s$) | 13,91 | 10,47 |
| $t_{trap}$ naar boven (Tobias) (in $s$) | 13,54 | 9,68 |
| $t_{trap}$ naar beneden (Tobias) (in $s$) | 12,56 | 9,31 |
| $t_{trap}$ naar beneden (Tobias) (in $s$) | 12,82 | 9,88 |
| $n_{treden}$ | 21 | 16 |
| β (in $s/trede$) | 0,62 | 0,63 |

*Tabel 3: metingen traplopen*

Door het gemiddelde te nemen van de looptijd per trede van trap 1 en trap 2 kan de looptijd per trap geschat worden op $0,63\ s/trede$.

$$\beta = 0,63\ s/trede \tag{38}$$

### 2.3.3 Bijzondere gewichten

Op de zijde van de glijbaan zijn de volgende metingen gedaan:

| | $t$ (in $s$) |
|---|---|
| Tobias | 14,76 |
| Tobias | 12,88 |
| Dylan | 12,25 |
| Dylan | 12,26 |

*Tabel 4: metingen glijbaan*

Door het gemiddelde te nemen van de metingen kan worden bepaald dat het gewicht van de zijde waarop de glijbaan ligt, $13,04\ s$ is. $w((85,\ 7)) = 13,04\ s$.

Op de zijde van de ingang van de fietsenkelder zijn de volgende metingen gedaan:

| | $t$ (in $s$) |
|---|---|
| Tobias | 20,54 |
| Tobias | 19,95 |
| Dylan | 19,76 |
| Dylan | 19,89 |

*Tabel 5: metingen ingang fietsenkelder*

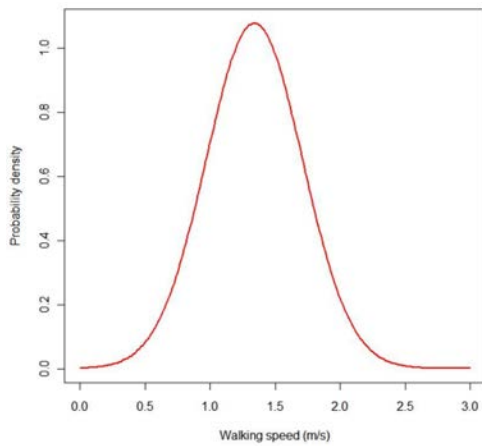
Door het gemiddelde te nemen van de metingen kan worden bepaald dat het gewicht van de zijde waarop de in- en uitgang van de fietsenkelder ligt, $20,04\ s$ is. $w(\{14,\ 3\}) = 20,04\ s$.

### 2.3.4 Loopsnelheden

De loopsnelheid $v$ van een mens (in $m/s$) is normaal verdeeld met een gemiddelde van $1,34\ m/s$ en een standaarddeviatie van $0,37\ m/s$ . (Buchmüller & Weidmann, 2006)

$$\sigma_v = 0,37\ m/s \tag{39}$$

$$\mu_v = 1,34\ m/s \tag{40}$$



*Figuur 14: normaalverdeling loopsnelheid (Sahaleh et al., 2012).*

Ter bepaling van de gewichten zijn de zijden opgemeten en omgerekend met de gemiddelde loopsnelheid $v_{\mu} = 1,34\ m/s$ naar looptijden. Later kan er, in de interface van het systeem, een loopsnelheid gekozen worden. Met de volgende formule kan een gecorrigeerde looptijd (zie formule 33) van een route $t_{cor}$ (in $s$) omgezet worden naar een de tijd $t_{gkz}$ (in $s$), op basis van een gekozen snelheid $v_{inp}$ (in $m/s$):

$$t_{gkz} = t_{cor} \cdot \frac{v_{\mu}}{v_{inp}} \tag{41}$$

## 2.4 De gewogen graaf

De afstanden van de zijden in de graaf worden opgemeten en omgezet naar looptijden. Hierbij zijn, zo nodig, de constanten λ en β (formule 36 en 38) opgeteld en de gewichten compleet gemaakt. Deze gewichten zijn, evenals de bijzondere gewichten, geplaatst in de ongewogen graaf. Hiermee is de gewogen graaf van de school gemaakt. Deze is te vinden in bijlage C.

# 3 Kortste pad programma

Een navigatiesysteem vereist een implementatie in code, waarvoor in dit onderzoek Python is gebruikt. In dit hoofdstuk wordt het ontworpen systeem uiteengezet. De link naar de GitHub-pagina, waar alle rauwe code staat, is te vinden in bijlage E. In dit hoofdstuk worden alleen de belangrijke gedeelten van het kortste pad programma uitgelicht.

## 3.1 Bestandsbeheer graaf

In het vorige hoofdstuk is de structuur van de graaf van het Hyperion Lyceum ontworpen en vervolgens zijn hier gewichten aan toegevoegd. De structuur en gewichten van deze gewogen graaf moeten overgebracht worden naar een bestand dat bruikbaar is voor een Python-code, hier wordt de eerder genoemde adjacency list (zie subparagraaf 1.1.4) voor gebruikt.

Om te beginnen is de getekende gewogen graaf (te zien in bijlage C) omgezet in een adjacency list in Google Spreadsheets. De link naar deze spreadsheet staat in bijlage E. Een Google Spreadsheet is makkelijk deel- en bewerkbaar, evenals georganiseerd. Dit is vervolgens gedownload in een csv-bestand zodat het met een converteercode omgezet kan worden naar een adjacency list binnen Python.

## 3.2 Code Dijkstra’s algoritme

Nu de graaf in de juiste vorm/layout staat, kan Dijkstra’s kortste pad algoritme erop losgelaten worden. De code hiervan is hieronder weergeven:

```
def check_connected_nodes(graph, node, path_weight, previous_node, visited):
    for connected_node, weight in graph[node]:
        if path_weight[node] + weight < path_weight[connected_node]:
            path_weight[connected_node] = path_weight[node] + weight
            previous_node[connected_node] = node
    visited[node] = True

def closest_unvisited_node(path_weight, visited):
    unvisited_distances = []
    for index, weight in enumerate(path_weight):
        if visited[index] == False :
            unvisited_distances.append((weight, index))
    if unvisited_distances:
        return min(unvisited_distances, key=lambda x: x[0])[1]

def find_route(startnode, endnode, previous_node):
    route = []
    current_node = endnode
    while current_node != startnode:
        route.append(current_node)
        current_node = previous_node[current_node]
    route.append(current_node)
    route.reverse()
    return route

def walking_time(tsys, speed, fastest_path):
    dtgem = 0.95
    tcor = tsys - float((len(fastest_path)-2)*dtgem)
    tgkz = round((tcor * (1.34/speed)),2)
```

```
    return tgkz

def run_algorithm(graph, startnode, endnode, speed):
    visited = [False] * len(graph)
    path_weight = [float('inf')] * len(graph)
    path_weight[startnode] = 0
    previous_node = [None] * len(graph)
    while not visited[endnode]:
        check_connected_nodes(graph, closest_unvisited_node(path_weight, visited), path_weight,
previous_node, visited)
    fastest_path = find_route(startnode, endnode, previous_node)
    return fastest_path, walking_time(path_weight[endnode], speed, fastest_path)
```

Deze code is opgedeeld in 5 functies, waarbij de laatste functie de eerdere samenbrengt.

De *check_connected_nodes* functie heeft de volgende invoer nodig: de graaf, een gekozen vertex, een lijst met lengtes van de (voorlopig) kortste routes naar de vertices, een lijst van de vorige vertex in deze routes en een lijst met welke vertices wel of niet bezocht zijn. Deze bepaalt vervolgens of de aangesloten vertices van de gekozen vertex, kortere routes kunnen krijgen die via de gekozen vertex gaan. Op basis hiervan worden de lijsten van de kortste routes en de vorige vorige vertices wel of niet aangepast. Ook wordt de gekozen vertex als bezocht gemarkeerd.

De *closest_unvisited_node* functie heeft slechts de lijst van bezochte vertices en de lijst van de lengtes van de kortste routes nodig. Deze bepaalt vervolgens wat de (voor het startpunt) dichtstbijzijnde vertex is die nog niet bezocht is. Dit is dus ook de output.

De *find_route* functie reconstrueert het pad van het startpunt naar het eindpunt. Hiervoor moet deze natuurlijk weten welke vertices precies het startpunt en eindpunt zijn. Ook heeft deze de lijst met vorige vertices in de kortste routes nodig. De functie geeft uiteindelijk een lijst terug met de vertices die de kortste route van het startpunt naar het eindpunt vormen, inclusief start- en eindpunt.

De *walking_time* functie trekt $dt_{gem}$ (formule 34) af van een gegeven tijd voor het aantal vertices en geeft de mogelijkheid om de de gecorrigeerde tijd te veranderen met een gekozen snelheid. Dit wordt gedaan aan de hand van formules 33 en 41.

Uiteindelijk brengt de *run_algorithm* functie alles samen. Deze heeft een graaf, een start- en eindpunt en een snelheid nodig. Als eerst initialiseert deze functie alle eerder genoemde lijsten (zoals afstand tot het startpunt). Vervolgens roept deze de *check_connected_nodes* functie aan voor de dichtstbijzijnde vertex, gevonden door de *closest_unvisited_node* functie. Dit wordt herhaald totdat het eindpunt bezocht is. Vervolgens geeft deze de route (gevonden door de *find_route* functie) en de lengte van deze route (aangepast door de snelheid) in een tupel(geordende lijst van twee elementen) terug.

Stel dat men de snelste route wil vinden van de fietsenkelder naar wiskundelokaal 44, met gemiddelde snelheid. Dat is van vertex 3 naar vertex 133. Dan geeft Dijkstra's algoritme de volgende output:

```
Route: [3, 4, 36, 37, 44, 38, 81, 80, 110, 111, 143, 142, 140, 138, 135, 134, 133], Time: 133.06
```

# 4 Casus s,t-stroom

Als opstap naar het verwerken van voetgangersstromen door de gehele graaf van de school, wordt eerst gemodelleerd aan de hand van een simpele casus. Deze casus houdt in dat er een stroom van 100 mensen van lokaal 1, naar lokaal 10 gaat. Het doel is om een Nash-stroom te vinden en te onderzoeken wat voor effect deze heeft op de gewichten van de graaf.

## 4.1 Vertragingsfunctie

Eerst moet er een vertragingsfunctie worden ontworpen voor de zijden binnen de graaf.

Voor de simpliciteit wordt de afstand $y$ (in $m$) van een zijde beschouwd als een direct product van de gemiddelde loopsnelheid $v_\mu = 1,34\ m/s$ en het berekende gewicht $w$ van een zijde (in $s$):

$$y = v_\mu \cdot w \tag{42}$$

Strikt genomen speelden constanten λ (formule 36) en β (formule 38) nog een rol in het gewicht, maar dit wordt buiten beschouwing gelaten.

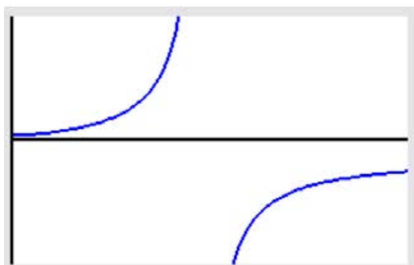
Ook zijn de gemiddelde loopsnelheid $v_\mu$ (formule 40) en de ongehinderde loopsnelheid $v_f$ (formule 31) gelijk aan elkaar:

$$v_f = v_\mu \tag{43}$$

Door middel van de formules 26, 28, 42 en 43 kan de volgende formule afgeleid worden, waarbij vertraging $L_e$ (in $s$) als functie van het aantal mensen $n$ (in $P$) genomen wordt:

$$L_e(n) = \frac{w}{1 - e^{-\gamma\left(\frac{x\,w\,v_f}{n} - \frac{1}{k'_{jam}}\right)}} \tag{44}$$

Hieronder is een grafiek van deze formule te zien:



*Figuur 15: grafiek van vertragingsfunctie 44, met n op de x-as en $L_e$ op de y-as (w=10, x=2 en de rest van de constanten genomen van Weidmann (1993)), (ti84calc, z.d.)*

In figuur 15 is te zien dat de formule hyperbolisch is. Uit de formule valt af te leiden dat de verticale asymptoot op $n = x\,w\,v_f\,k'_{jam}$ ligt. Daarna wordt $L_e$ negatief, dit is in strijd met het

idee van een vertragingsfunctie (formule 14). Ook is $L_e$ ongedefinieerd voor $n = 0$. Vanwege deze “tekortkomingen” moet een stuksgewijs gedefinieerde functie opgesteld worden als vertragingsfunctie:

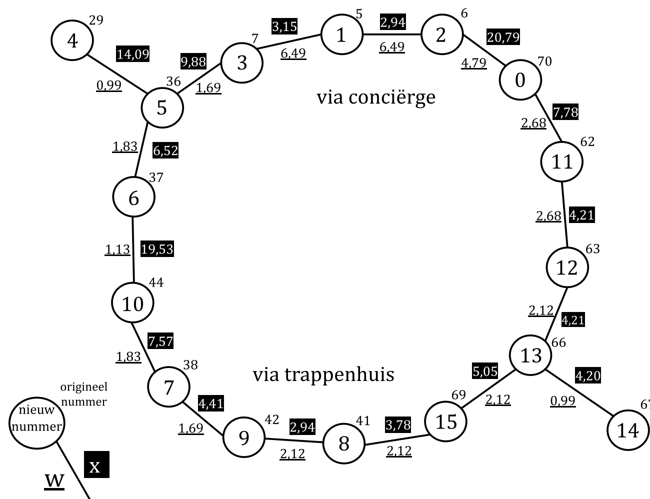
$$L_e(n) = \begin{cases} w & als \quad n = 0 \\ \dfrac{w}{1 - e^{-\gamma\left(\frac{x\, w\, v_f}{n} - \frac{1}{k'_{jam}}\right)}} & als \quad 0 < n < x\, w\, v_f\, k'_{jam} \\ \infty & als \quad n \geq x\, w\, v_f\, k'_{jam} \end{cases} \tag{45}$$

Met deze formule kan voor een zijde een dynamisch gewicht worden berekend, afhankelijk van het aantal mensen op die zijde.

# 4.2 Stromingsmodel

## 4.2.1 De deelgraaf

Voor deze casus is een deelgraaf van de totale graaf van het Hyperion Lyceum opgesteld:




*Figuur 16: deelgraaf casus*

Belangrijk om te onthouden is dat er nieuwe vertexnummers zijn gegeven aan de vertices in deze deelgraaf. De nummers in deze deelgraaf staan dus los van die in de originele graaf, ontwikkeld in hoofdstuk 2. Ook staan ze los van de gebruikte lokaalnummers: vertex 4 is lokaal 1 en vertex 14 is lokaal 10.

Elke zijde krijgt hier twee gegevens toegewezen: een breedte $x$ en het gewicht $w$ (de onbelemmerde, gemiddelde looptijd). In het conversie proces worden, naast de breedte, nog twee dynamische grootheden aan de connecties in de adjacency list toegevoegd: een aantal voetgangers $n$ op de zijde en een vertraging $L_e$ (het dynamische gewicht). Een verbinding in een adjacency list van deze graaf ziet er als volgt uit:

$$u \in V : [v \in V,\ w(\{u, v\}),\ x,\ n,\ L_e] \qquad (46)$$

### 4.2.2 Verantwoording

In het theoretisch kader, inzake stromen (paragraaf 1.3), werd gekozen voor een continue stroom omdat een individuele deelnemer weinig invloed heeft. Op het Hyperion Lyceum zitten 859 leerlingen (Scholen op de kaart, z.d.) die in veel kleinere deelstromen door de gangen van de school bewegen. Hierdoor is gekozen voor een discrete benadering.

Elke leerling binnen een verkeersstroom is tijd- en plaatsgebonden. Met een continue stroom wordt dit verwaarloosd. Echter is er gekozen voor een discrete stroom, dus moet hier rekening mee gehouden worden. Voor deze tijd- en plaatsgebondenheid is de Python-functie *Asyncio* essentieel. Deze stelt de programmeur in staat asynchroon taken te laten draaien: meerdere taken kunnen tegelijk aan de gang zijn. Zo kan de locatie van elke leerling over de tijd bijgehouden worden. Wel betekent dit dat het draaien van een model met werkelijke tijd werkt: als de code een taak laat slapen voor 20 seconden, slaapt deze ook echt voor 20 seconden. Wel kan de tijd sneller afgespeeld worden door met een variabele *Deltatime* te vermenigvuldigen in elke slaapfunctie. Een *Deltatime* van 0,1 bijvoorbeeld, zou de tijd met factor 10 versnellen.

Ook is er gekozen om een Nash-stroom te modelleren. Dit volgt uit de aanname dat de leerlingen zo snel mogelijk bij de les willen zijn en geïnformeerd zijn over congestie op mogelijke paden. Door de tijd- en plaatsgebondenheid wordt het vinden van de Nash-stroom iets anders aangepakt: op het moment van vertrek van een leerling wordt bepaald welke van de twee paden het snelst is. Dit zou ervoor zorgen dat de stroom een Nash-stroom nadert en in veel gevallen niet exact behaalt: Het behalen van een exacte Nash-stroom vereist een gelijkstelling van twee of meer vertragingsfuncties (zie voorbeeld in subparagraaf 1.3.2). Met een vertragingsfunctie zoals formule 45 zou in veel gevallen de oplossing bestaan uit irrationale getallen, wat natuurlijk niet in discrete aantallen uitgedrukt kan worden.

### 4.2.3 Codering

Hierbij kan het volgende algoritme gecodeerd worden:

```
import asyncio
import csv
from dijkstra_casus import run_algorithm
from asyncio import Lock

latency_lock = Lock()

def latency_function(x, w, n):
    if n > 0:
        kjam = 5.4
        vf = 1.34
```

```
        e = 2.718281828
        gamma = -1.913
        L = w / (1 - (e ** (gamma * (((x * w * vf) / n) - (1 / kjam)))))
        if L < 0:
            L = float('inf')
    else:
        L = w
    return L

async def walk_route(path, graph, deltatime):
    for i, node in enumerate(path):
        if i < len(path) - 1:
            next_node = path[i + 1]
            async with latency_lock:
                for index_connection, connection in enumerate(graph[node]):
                    if connection[0] == next_node:
                        connection[3] += 1
                        w = connection[1]
                        x = connection[2]
                        Le = latency_function(x, w, connection[3])
                        connection[4] = Le
                        store_index_connection = index_connection
                        break
                undirected = False
                for index_connection, connection in enumerate(graph[next_node]):
                    if connection[0] == node:
                        connection[3] += 1
                        connection[4] = latency_function(x, w, connection[3])
                        store_index_reverse_connection = index_connection
                        undirected = True
                        break
            await asyncio.sleep(Le*deltatime)
            async with latency_lock:
                graph[node][store_index_connection][3] -=1
                Le_after = latency_function(x, w, graph[node][store_index_connection][3])
                graph[node][store_index_connection][4] =  Le_after
                if undirected:
                    graph[next_node][store_index_reverse_connection][3] -=1
                    graph[next_node][store_index_reverse_connection][4] = Le_after

async def flow(s, t, m, deltatime, tleave, graph):
    tasks = []
    for n in range(1, m + 1):
        path = run_algorithm(graph, s, t, 1.34)[0]
        task = asyncio.create_task(walk_route(path, graph, deltatime))
        tasks.append(task)
        await asyncio.sleep(tleave * deltatime)
    await asyncio.gather(*tasks)


async def main(s, t, m, deltatime, tleave, graph):
    await asyncio.gather(
        flow(s, t, m, deltatime, tleave, graph),
    )
```

In de eerste functie wordt formule 45 en constanten 29, 30 en 31 gecodeerd.

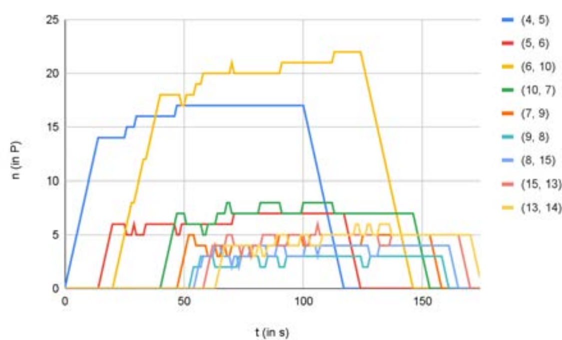
In de tweede (async)functie wordt het lopen van een pad van één persoon gemodelleerd. Voor elke zijde waar deze op loopt wordt aan het begin zijn aanwezigheid opgeteld bij $n$ en hiermee $L_e$ opnieuw berekend. Dan slaapt de taak voor de duur van $L_e$. Ten slotte wordt zijn aanwezigheid bij n afgetrokken en wederom $L_e$ opnieuw berekend en gaat deze persoon door naar de volgende zijde van zijn pad.

In de derde (async)functie wordt het vertrek uit een klaslokaal gesimuleerd: voor elke persoon wordt het snelste pad berekend met Dijkstra's algoritme. Vervolgens wordt deze op pad gestuurd door de tweede (async)functie te draaien. Vervolgens slaapt deze taak één seconde; het duurt dus een seconde voor één persoon om de deur uit te gaan. Dit wordt gedaan voor alle 100 leerlingen.

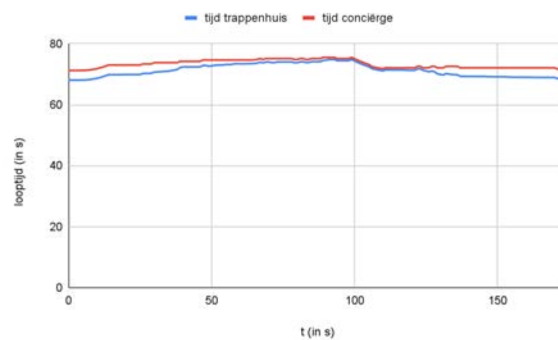
In de vierde functie (async)functie wordt het programma gestart.

## 4.3 Resultaten

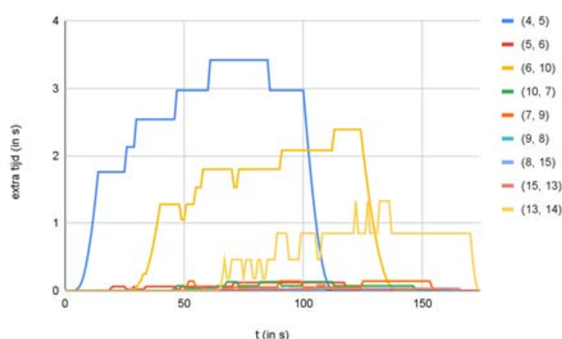
Door een verkeersstroom van 100 mensen van lokaal 1 (vertex 4) naar lokaal 10 (vertex 14) te simuleren, voor het gemak wordt $(4, 14, 100)$ genoteerd, kunnen de volgende resultaten verkregen worden:




*Figuur 17: personenaantallen op zijden, (4,14,100)*



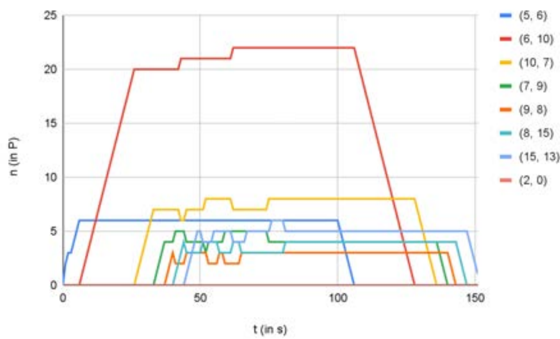

*Figuur 18: looptijden paden, (4,14,100)*



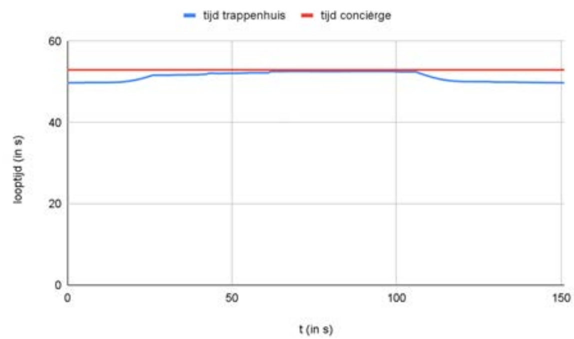

*Figuur 19: extra tijd op zijden, (4,14,100)*

In figuur 17 is te zien dat er geen personen via de conciërge zijn gelopen. Er is immers geen lijn die aangeeft dat er personen over de zijden van dit pad zijn gelopen. Toch is te zien in figuur 18 dat de looptijd van het pad via de conciërge omhoog gaat. Dit valt te verklaren door het feit dat de meeste extra tijd door congestie ontstaat op de twee overlappende zijden van de twee paden. Hierdoor stijgt ook het pad via de conciërge in looptijd terwijl er niemand overheen loopt, wat er weer voor zorgt dat via het trappenhuis altijd sneller is en er niemand via de conciërge loopt. In figuur 19 is het logische resultaat te zien dat de zijden met de kleinste breedte $x$ een hogere vertraging krijgen door congestie.

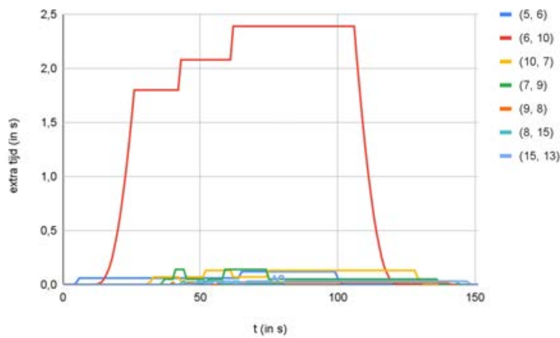
Hieruit volgde de vraag of bij uitsluiting van de minst brede, overlappende zijden $\{4, 5\}$ en $\{13, 14\}$ wel mensen via de conciërge zouden gaan. Voor een stroom van vertex 5 naar vertex 13, waardoor deze zijden uitgesloten worden, is het volgende resultaat verkregen:




*Figuur 20: personenaantallen op zijden, (5,13,100)*




*Figuur 21: looptijden paden, (5,13,100)*




*Figuur 22: extra tijd op zijden, (5,13,100)*

Ook in deze stroom gaat iedereen via het trappenhuis. In figuur 21 is te zien dat de looptijd via het trappenhuis altijd onder de looptijd via de conciërge blijft, alhoewel de laatste (in tegenstelling tot $(4, 14, 100)$) constant blijft. Dit valt te verklaren door het feit dat het uitsluiten van de minst brede zijden ervoor zorgt dat de looptijd via het trappenhuis veel minder snel oploopt. Dit is in figuur 22 is te zien: alleen de extra tijd op zijde $(6, 10)$, wat het trappenhuis zelf is, wordt significant hoog.

## 4.4 Uitbreidmogelijkheden en tekortkomingen

De resultaten van het stroommodel zijn allemaal vrij logisch: de smalste gangen ervaren de meeste extra vertraging, de personenaantallen gaan omhoog en omlaag wanneer dat zou moeten evenals de totale looptijd. Het is dus goed mogelijk om dit uit te breiden en door de gehele school toe te passen.

Echter zijn er twee tekortkomingen in het model waarmee rekening gehouden moet worden:

- Het model kan niet goed omgaan met dichtheden, hoger dan de kritieke dichtheid $k'_{krit}$. Dit komt doordat file niet van één zijde op een ander kan overslaan, wat in de werkelijkheid wel gebeurt. Als er op een trap zware congestie is, zal er op de gang vóór de trap ook congestie zijn. Als deze trap en gang gerepresenteerd worden door twee verschillende zijden in het model, en er ontstaat zware congestie op de zijde van de trap, dan zal dit niet doorgegeven worden aan de zijde van de gang. Op de gang zal alles doorlopen met een niet-verminderende intensiteit waardoor de trap alsmaar voller wordt, soms hoger dan de maximale dichtheid $k'_{jam}$ waardoor uit formule 45 volgt dat deze mensen oneindig lang op deze zijde zullen zijn.
- Dijkstra's algoritme kijkt alleen naar de gewichten op de zijden op het moment van vertrek. Het algoritme berekent het snelste pad op basis van de gewichten van de zijden op het moment van vertrek van een persoon, in plaats van op basis van hoe de gewichten zullen zijn op het moment dat deze persoon zich ook echt op die zijde bevindt. Dit heeft alleen effect op de eerste en laatste mensen die de deur uit gaan aangezien voor hen de gewichten anders zijn als ze vertrekken dan wanneer ze echt op die zijde zijn.

# 5 Stroommodel

De volgende stap is om de gehele verkeersstroom in het Hyperion Lyceum te modelleren. Hierbij wordt de gemaakte code en verkregen kennis uit de casus op grotere schaal toegepast.

## 5.1 Dubbel Gewogen Graaf

Zoals de deelgraaf uit de casus moet er een dubbel gewogen graaf voor de gehele school gemaakt worden. Dit is gedaan door simpelweg de breedte van de gang van een zijde op te meten. Als deze meerdere breedtes heeft, is de minimale breedte genomen als breedte $x$. Deze graaf met breedtes is te vinden in bijlage D.

## 5.2 Code

Voor de modellering van de gehele verkeersstroom van leerlingen door het Hyperion Lyceum is de code uit de casus genomen en deze aangepast opdat deze niet een stroom van één lokaal naar één ander lokaal stuurt, maar van één lokaal naar meerdere andere lokalen. Ook zodat deze even wacht totdat het lokaal open gaat. Daarnaast is een nieuwe code geschreven om meerdere van deze stromen te starten:

```
from trafficflow import outward_flow
from converter_traffic import csv_to_adjacency_list
import random
import asyncio
import csv
from asyncio import Lock
from scipy.stats import beta
import os
import pandas
import copy
from asyncio import Lock

def divide(m, period, starting_rooms, ending_rooms):
    capacity_starting_rooms = [[room, max_capacity, 0] for room, max_capacity in starting_rooms]
    capacity_ending_rooms = [[room, max_capacity, 0] for room, max_capacity in ending_rooms]
    students = 0
    division = {room : [0,[]] for room, max_capacity in starting_rooms}
    while students < m:
        index_starting_room = random.randint(0, len(capacity_starting_rooms)-1)
        index_ending_room = random.randint(0, len(capacity_ending_rooms)-1)
        if capacity_starting_rooms[index_starting_room][2] <
capacity_starting_rooms[index_starting_room][1] :
            if capacity_ending_rooms[index_ending_room][2] < capacity_ending_rooms[index_ending_room][1]:
                capacity_starting_rooms[index_starting_room][2] += 1
                capacity_ending_rooms[index_ending_room][2] += 1
                students += 1

division[capacity_starting_rooms[index_starting_room][0]][1].append(capacity_ending_rooms[index_ending_room
][0])
            else: capacity_ending_rooms.pop(index_ending_room)
        else: capacity_starting_rooms.pop(index_starting_room)
    for room in division:
        division[room][0] = float(round((beta.rvs(2,2))*period, 2))
    return division
```

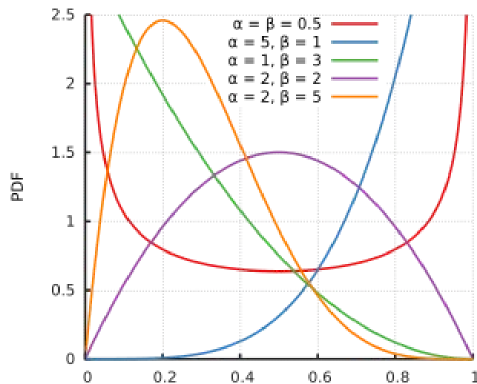
```
async def run_model(graph_lock, graph, m, tleave, deltatime, period, starting_rooms, ending_rooms,
timeframe, model_nr):
    tasks = []
    division = divide(m, period, starting_rooms, ending_rooms)
    for room, div in division.items():
        task = asyncio.create_task(outward_flow(graph_lock, div[0], room, div[1], graph, tleave,
deltatime))
        tasks.append(task)
    output_file = './traffic_total/output.csv'
    if not os.path.exists(output_file):
        with open(output_file, 'w', newline='') as csv_file:
            csv_writer = csv.writer(csv_file)
            csv_writer.writerow(['t'])
            for i in range(1, timeframe + 1):
                csv_writer.writerow([i])
    editable_output_file = pandas.read_csv(output_file)
    busiest_graph = await indicator(graph_lock, graph, deltatime, model_nr, timeframe,
editable_output_file)
    for task in tasks:
        task.cancel()
    editable_output_file.to_csv(output_file, index=False)
    print(f'model nr. {model_nr} is done running')
    return busiest_graph

async def indicator(graph_lock, graph, deltatime, model_nr, timeframe, editable_output_file):
    current_highest_total_people = 0
    async with graph_lock:
        busiest_graph = copy.deepcopy(graph)
    t = 0
    column = []
    while t < timeframe:
        t+=1
        print(t, end=',')
        total_people = 0
        async with graph_lock:
            for room, connections in graph.items():
                for connection in connections:
                    total_people += connection[3]
        total_people /= 2
        if(total_people > current_highest_total_people):
            current_highest_total_people = total_people
            async with graph_lock:
                busiest_graph = copy.deepcopy(graph)
        column.append(total_people)
        await asyncio.sleep(deltatime)
    editable_output_file[model_nr] = column
    return busiest_graph

async def monte_carlo(graph_file, m, tleave, deltatime, period, starting_rooms, ending_rooms, timeframe,
sample_amount):
    graph_lock = Lock()
    sample_graphs = []
    for model_nr in range(1, sample_amount + 1):
        graph = csv_to_adjacency_list(graph_file)
        sample_graph = await run_model(graph_lock, graph, m, tleave, deltatime, period, starting_rooms,
ending_rooms, timeframe, model_nr)
        sample_graphs.append(sample_graph)
    monte_carlo_graph = {}
    for node, edges in sample_graphs[0].items():
        mc_edges = []
        for index, edge in enumerate(edges):
            sum_weight = sum(sampled_graph[node][index][4] for sampled_graph in sample_graphs)
            mc_weight = round(sum_weight / len(sample_graphs), 2)
            mc_edges.append([edge[0], mc_weight])
        monte_carlo_graph[node] = mc_edges
```

```
    return monte_carlo_graph
```

In de *divide* functie wordt, voor elke leerling, het beginpunt en het eindpunt uniform willekeurig (dus alles heeft een gelijke kans) gekozen zolang de lokalen hiervoor de capaciteit hebben. De meeste lokalen hebben bijvoorbeeld een capaciteit van 32 leerlingen. Het moment waarop een lokaal opengaat volgt ook een willekeur in bepaalde mate. Deze wordt gekozen aan de hand van een parabolische bètaverdeling: $beta(2, 2)$. Er is gekozen voor een bètaverdeling omdat er enige correlatie zit in het moment waarop de deuren van een lokaal opengaan en een bètaverdeling een begrensd ondersteuningsinterval heeft (bij een normaalverdeling bestaat bijvoorbeeld de kans dat de deur 10000 uur later open gaat).




*Figuur 23: verschillende bètaverdelingen (Wikipedia, z.d.)*

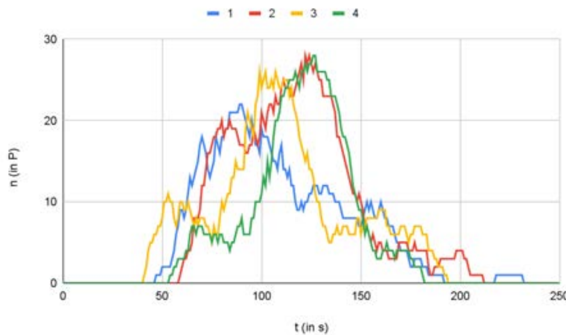
De *run_model* functie start de *divide* functie en gebruikt de waarden hieruit om voor elk lokaal de *outward_flow*[5] functie te starten. Deze functies samen vormen dus in principe het model. Echter geeft het geen enkele data terug over wat er precies gebeurt. Dit wordt dus geregeld in de *indicator* functie, deze houdt het totaal aantal mensen op de graaf bij en slaat de graaf op wanneer dit aantal een maximum bereikt. Deze *indicator* functie kan ook het model stoppen wanneer het tijdsdomein waarover gemeten wordt voorbij is. Want het laten draaien van het model als er geen data van verkregen wordt, heeft geen zin. Bovendien is er een mogelijkheid dat bepaalde taken oneindig duren (reeds besproken in paragraaf 4.4) en deze dus afgebroken moeten worden.

Één graaf kan echter helemaal geen goede representatie zijn van de file die doorgaans ontstaat. Er wordt toeval gebruikt in het kiezen van de begin- en eindpunten van leerlingen, evenals het moment waarop een deur opengaat. Zo kan het zijn dat één graaf irreguliere resultaten laat zien. Het is dus beter om een gemiddelde te nemen van dit toevalsproces. Daarom regelt de *monte_carlo* functie ervoor dat het model zich een gekozen aantal keer herhaalt en een gemiddelde neemt van de opgeslagen grafen.

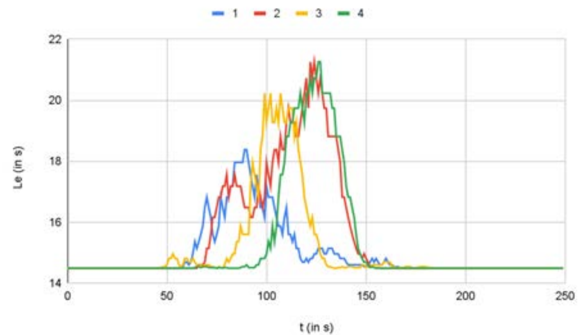
[5] Deze functie is een aangepaste versie van de flow-functie, gebruikt in de casus.

## 5.3 Resultaten

Het visualiseren van congestie door de gehele school is erg lastig, er is dus gekozen om de congestie op één zijde te visualiseren. Dit is zijde $\{38, 81\}$ met $w = 14,49\ s$ en $x = 1,13\ m$ en weergeeft de zuidoostelijke trap, van de eerste naar de tweede verdieping. Het model heeft vier keer gedraaid. Hierbij is een totaal van 859 leerlingen ingevoerd, een periode van deuren openen van 120 seconden ingevoerd (dit is het begrensde ondersteuningsinterval van de eerder genoemde bètaverdeling) en alle lokalen als begin- en eindpunten genomen. Hierbij zijn de volgende resultaten verkregen:




*Figuur 24: aantal personen op {38, 81}*




*Figuur 25: looptijd op {38, 81}*

In figuur 24 is te zien dat in alle vier de simulaties de personenstroom ongeveer op hetzelfde moment start en eindigt. Echter liggen de maxima nog vrij ver uit elkaar qua tijd. Bovendien, door de hyperbolische aard van formule 45, worden de verschillen in maximumhoogte in figuur 24 extra versterkt in figuur 25. In hogere druktes kunnen enkele personen namelijk veel hogere vertragingen veroorzaken dan in lagere druktes. Deze grote verschillen in maximale vertragingen in figuur 25, benadrukken de noodzaak van het nemen van een gemiddelde van meerdere grafen.
Bij verdere analyse van 'tussentijdse' grafen van meerdere simulaties is te zien dat af en toe een zijde (meestal $\{142, 143\}$) op oneindig uitkomt. Dit is uiteraard het resultaat van het eerder besproken probleem dat het model niet goed kan omgaan met zware congestie.

## 5.4 Toepassing

De congestiegrafen (grafen waarin congestie is meegenomen) worden geïmplementeerd in de navigator applicatie (meer hierover in het volgende hoofdstuk). Er worden drie congestiegrafen genomen:

- Tussen twee lessen in: iedereen loopt van lokaal naar lokaal.
- Tussen een les en een pauze: iedereen loopt van lokaal naar een pauzeplek (zoals de kantine) of andersom.
- Aan het begin van de dag: iedereen loopt van de ingangen van de school naar de les.

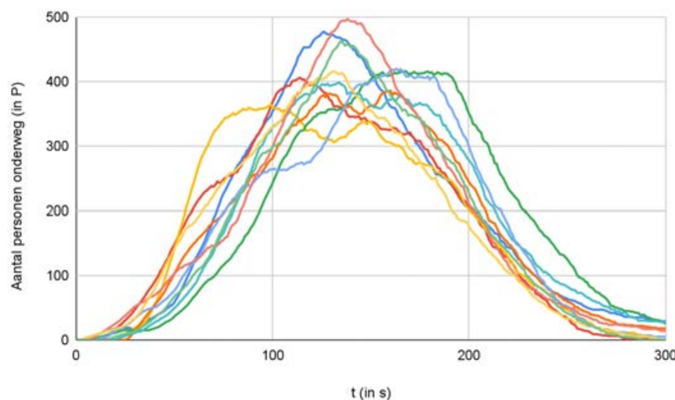
Daarnaast geldt op momenten zonder drukte simpelweg de eerder berekende standaardgraaf (graaf zonder file). Deze wordt *graph_normal* genoemd. De namen van de aparte congestiegrafen volgen nog. Alle congestiegrafen zijn te vinden op de GitHub-pagina, waarvan de link te vinden is in bijlage E.

### 5.4.1 Van lokaal naar lokaal

Voor het modelleren van de voetgangersstroom tussen twee lessen in, wordt de volgende invoer gebruikt:

```
starting_rooms = [(28, 32), (29, 32), (30, 32), (32, 32), (39, 32), (40, 32), (45, 32), (47, 32), (48, 32),
(52, 32), (67, 32), (68, 32), (92, 32), (89, 32), (88, 32), (87, 32), (78, 32), (79, 32), (75, 32), (74,
32), (96, 32), (100, 32), (103, 32), (104, 32), (112, 32), (113, 32), (115, 32), (116, 32), (133, 32),
(134, 32), (135, 32), (136, 32), (146, 32), (147, 32)]
ending_rooms = starting_rooms
period = 180
m = 859
tleave = 1
timeframe = 300
sample_amount = 10
```

Voor de begin- en eindpunten worden simpelweg alle lokalen genomen, met elk een capaciteit van 32 mensen. De periode waarin deuren open kunnen gaan (bèta verdeeld) is 180 seconden. In totaal lopen er 859 personen. De tijd voor één leerling om de deur uit te gaan in het model is één seconde. Er wordt gemeten over 300 seconden. Voor de congestiegraaf wordt een gemiddelde van 10 grafen genomen, het model draait dus 10 keer. Ten gevolge van deze invoer, zijn de volgende resultaten gevonden:




*Figuur 26: personenaantallen over de tijd*

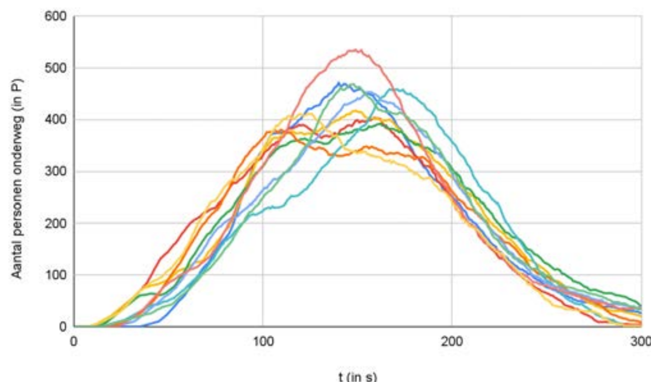
In de figuur is te zien dat de tien verschillende herhalingen van het model een relatief gelijk verloop hebben inzake het aantal lopende mensen over de tijd.
Voor $t = 0$ wordt hier 5 minuten voor het einde van de les genomen, dat is het moment dat de deuren in principe open moeten gaan (schoolbeleid). Deze congestiegraaf gaat gebruikt worden tussen 60 en 240 seconden nadat de deuren open kunnen gaan, omdat op dit interval de personenaantallen hoog genoeg zijn om file te verwachten. Deze graaf krijgt de naam *class_to_class*.

## 5.4.2 Tussen les en pauze

Voor een simulatie van de verkeersstroom tussen een les en de pauze wordt de volgende invoer gebruikt:

```
starting_rooms = [(28, 32), (29, 32), (30, 32), (32, 32), (39, 32), (40, 32), (45, 32), (47, 32), (48, 32), (52, 32), (67, 32), (68, 32), (92, 32), (89, 32), (88, 32), (87, 32), (78, 32), (79, 32), (75, 32), (74, 32), (96, 32), (100, 32), (103, 32), (104, 32), (112, 32), (113, 32), (115, 32), (116, 32), (133, 32), (134, 32), (135, 32), (136, 32), (146, 32), (147, 32)]
ending_rooms = [(19, 46), (22, 46), (24, 46), (8, 46), (9, 46), (2, 46), (60, 46), (62, 46), (66, 46), (69, 46), (43, 46), (85, 46), (127, 46), (76, 46), (138, 46), (147, 46), (1, 46), (58, 46), (125, 46)]
period = 180
m = 859
tleave = 1
timeframe = 300
sample_amount = 10
```

Voor de startpunten worden de lokalen gebruikt. Voor de eindpunten zijn plekken genomen waar leerlingen vaak hun pauze doorbrengen. De capaciteiten van deze plekken zijn overal gelijk en tellen op tot (net iets boven) 859 personen zodat alle personen gelijk verdeeld worden over deze pauzeplekken. De overige invoer blijft gelijk aan wat eerder genoemd is. Deze invoer geeft de volgende resultaten:




*Figuur 27: personenaantallen over de tijd*

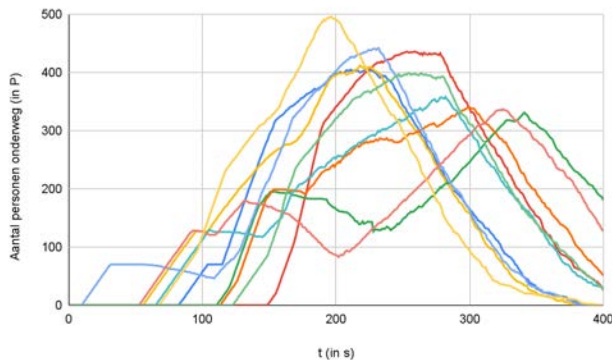
De personenaantallen over de tijd vertonen sterke gelijkenis met die van de vorige subparagraaf. Alle herhalingen hebben een relatief gelijk verloop. Echter verschillen de zijden waarop congestie ontstaat wel van die tussen lessen in. In dit geval wordt voor $t = 0$, drie minuten voor het begin van de pauze of vier minuten voor het einde van de pauze genomen. Deze congestiegraaf gaat eveneens gebruikt worden tussen 60 en 240 seconden na $t = 0$. Deze graaf krijgt de naam *class_to_break*. Er wordt aangenomen dat de congestie tussen een les en daarna een pauze, en een pauze en daarna een les hetzelfde is. Voor beide situaties wordt dus de graaf *class_to_break* gebruikt.

### 5.4.3 Start van de dag, 9:00

Voor de start van de dag is het volgende ingevoerd:

```
ending_rooms = [(28, 32), (29, 32), (30, 32), (32, 32), (39, 32), (40, 32), (45, 32), (47, 32), (48, 32),
(52, 32), (67, 32), (68, 32), (92, 32), (89, 32), (88, 32), (87, 32), (78, 32), (79, 32), (75, 32), (74,
32), (96, 32), (100, 32), (103, 32), (104, 32), (112, 32), (113, 32), (115, 32), (116, 32), (133, 32),
(134, 32), (135, 32), (136, 32), (146, 32), (147, 32)]
starting_rooms = [(3, 400), (11, 130), (21, 70)]
period = 240
m = 600
tleave = .3
timeframe = 400
sample_amount = 10
```

In dit geval worden de lokalen als eindpunten genomen. De fietsenkelder en twee ingangen aan de voor- en achterkant van de school worden als beginpunten genomen. Elk van deze punten heeft een capaciteit die bedoeld is om de verdeling te sturen: 400 mensen beginnen in de fietsenkelder, 130 aan de voorkant van de school en 70 aan de achterkant. De periode waar “de deuren open gaan” wordt wat groter gekozen, op 240 seconden. Niet iedereen begint om 9 uur ‘s ochtends (veel leerlingen beginnen later op de dag) dus worden er maar 600 mensen genomen. Er zitten nu nog maar 0,3 seconden tussen twee vertrekkende mensen omdat alle beginpunten meerdere en grotere uitgangen hebben. Dit zou eigenlijk moeten verschillen voor de drie uitgangen, aangezien er veel meer mensen vertrekken vanuit de fietsenkelder dan vanuit de ingangen, echter zou dit een totale verandering van de code vereisen. Door de grotere periode waarin de deuren open gaan wordt er gemeten over een duur van 400 seconden. Wederom worden er 10 herhalingen genomen. Uit deze invoer zijn de volgende resultaten verkregen:




*Figuur 28: personenaantallen over de tijd*

In de figuur is te zien dat de herhalingen een zeer verschillend verloop hebben. Dit valt te verklaren door het kleine aantal beginpunten met veel grotere capaciteiten, de grotere periode waarin de deuren open kunnen gaan en de lagere tijd tussen twee vertrekkende personen. Als een lokaal open gaat, schieten de personenaantallen in één keer omhoog. In de vorige subparagrafen ging dat veel gelijkmatiger. Deze congestiegraaf gaat gebruikt worden tussen 100 en 400 seconden na $t = 0$. Dat is dus van 8:55 tot 9:00. Deze graaf krijgt de naam *start_to_class*. In tegenstelling tot de vorige subparagraaf, wordt deze graaf alleen gebruikt aan het begin van de dag, niet aan het einde. Dit volgt uit de constatering dat er meer mensen om 9:00 beginnen dan dat er om 16:00 eindigen. Om 16:00 is er vrijwel nooit file.

### 5.4.4 Programmering in applicatie

In de volgende tabel zijn de gegevens waarop de congestiegrafen worden gebruikt, beschreven in de vorige subparagrafen, nog eens op een rijtje gezet:

| congestiegraaf | Wanneer is $t = 0$ in dit geval? | Hoeveel seconden na $t = 0$ wordt deze gebruikt? |
|---|---|---|
| class_to_class | Vijf minuten voordat de volgende les begint | Tussen 60 en 240 |
| class_to_break | Drie minuten voor het begin van de pauze en vier minuten voor het eind van de pauze. | Tussen 60 en 240 |
| start_to_class | 8:53:20 | Tussen 100 en 400 |

*Tabel 6: recollectie van tijden om congestiegrafen in te zetten*

Naast deze gegevens moet ook nog gekeken op welke momenten op een schooldag de verschillende congestiegrafen gebruikt moeten worden. Hiertoe moet gekeken worden naar het rooster op het Hyperion Lyceum:

| Tijden | Activiteit |
|---|---|
| 16:00 - 9:00 | geen school |
| 9:00 - 9:45 | 2e lesuur |
| 9:45 - 10:30 | 3e lesuur |
| 10:30 - 10:45 | 1e pauze (klein) |
| 10:45 - 11:30 | 4e lesuur |
| 11:30 - 12:15 | 5e lesuur |
| 12:15 - 12:45 | 2e pauze (groot) |
| 12:45 - 13:30 | 6e lesuur |
| 13:30 - 14:15 | 7e lesuur |
| 14:15 - 14:30 | 3e pauze (klein) |
| 14:30 - 15:15 | 8e lesuur |
| 15:15 - 16:00 | 9e lesuur |

*Tabel 7: rooster doorgaanse schooldag*

Van 8:15 tot 9:00 en van 16:00 tot 16:45 zijn in principe ook lesuren maar deze worden zelden gebruikt en worden daardoor buiten beschouwing gelaten.

Combineren van de twee bovenstaande tabellen geeft:

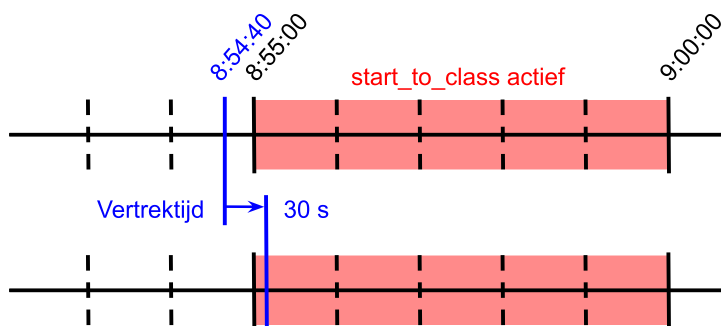
| Tijden | Graaf te gebruiken |
|---|---|
| 00:00 - 8:55 | normal |
| 8:55 - 9:00 | start_to_class |
| 9:00 - 9:41 | normal |
| 9:41 - 9:44 | class_to_class |
| 9:44 - 10:28 | normal |
| 10:28 - 10:31 | class_to_break |
| 10:31 - 10:42 | normal |
| 10:42 - 10:45 | class_to_break |
| 10:45 - 11:26 | normal |
| 11:26 - 11:29 | class_to_class |
| 11:29 - 12:13 | normal |
| 12:13 - 12:16 | class_to_break |
| 12:16 - 12:42 | normal |
| 12:42 - 12:45 | class_to_break |
| 12:45 - 13:26 | normal |
| 13:26 - 13:29 | class_to_class |
| 13:29 - 14:13 | normal |
| 14:13 - 14:16 | class_to_break |
| 14:16 - 14:27 | normal |
| 14:27 - 14:30 | class_to_break |
| 14:30 - 15:11 | normal |
| 15:11 - 15:14 | class_to_class |
| 15:14 - 16:00 | normal |

*Tabel 8: tijdsintervallen van congestiegrafen*

In de GUI kiest de gebruiker een tijd. Dit wordt de *time_input* genoemd. Daarna kiest hij/zij het *type_tijd*. De twee mogelijke types zijn vertrek en aankomst. Op basis van *time_input*, en het *type_tijd*, wordt de juiste graaf gekozen.

Als het *type_tijd* vertrek is, wordt 30 seconden opgeteld bij *time_input*. Het resultaat bepaalt de graaf die gebruikt wordt (*normal, class_to_class, class_to_break of start_to_class*). Dit wordt door de computer gedaan door in tabel 8 te kijken en de corresponderende graaf te selecteren. Als het *type_tijd* aankomst is, wordt 30 seconden afgetrokken van de ingevoerde tijd.

De reden voor het optellen/aftrekken van 30 seconden, is dat de drukte die heerst op de ingevoerde tijd (*time_input)*, niet per se de drukte zal zijn gedurende de hele route. Stel bijvoorbeeld dat van een route van 2 minuten, *time_input* 8:54:40 (op de seconde) is, en het *type_tijd* vertrek is. De graaf die bij *time_input* hoort, is volgens tabel 8 *normal*, dus compleet rustig. Maar het overgrote deel van de route zou de congestiegraaf *start_to_class* van toepassing moeten zijn. In dit geval zou het gebruik van de graaf *start_to_class* een beter beeld geven van de werkelijke reistijd dan de graaf *normal* die bij de *time_input* hoort (zie figuur 29 voor verduidelijking). Wanneer aankomst de gekozen *type_tijd* is, geldt een omgekeerde logica, er moet dus 30 seconden van de ingevoerde tijd afgetrokken worden. Er is gekozen voor 30 seconden, omdat het grofweg de helft van de gemiddelde reistijd is tussen twee willekeurige punten in het Hyperion Lyceum. Deze verschoven reistijd wordt exclusief gebruikt voor het bepalen van de juiste graaf en niet in verdere berekeningen meegenomen.



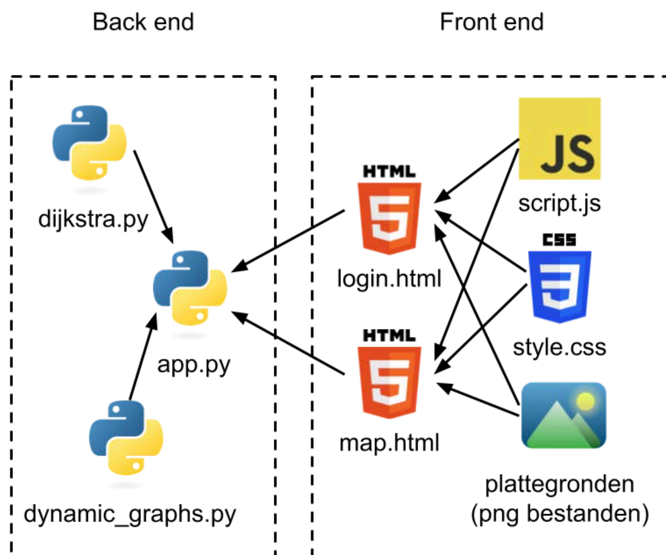

*Figuur 29: voorbeeld tijdverschuiving*

Het beschreven systeem is in een Python-functie gezet, dat gebruikt wordt in de gebruikersinterface. Deze functie heet *graph_for_time.* Het gebruik hiervan wordt beschreven in paragraaf 6.1.

# 6 Gebruikersinterface (GUI)

Voor de gebruikers is een website ontworpen, waarin al het werk dat voor dit profielwerkstuk is verricht, samenkomt. De website heet ook de gebruikersinterface, in het Engels de *graphical user interface*, kortweg GUI. De link naar de website is te vinden in bijlage E en screenshots van de website in bijlage F. Dit hoofdstuk legt de werking van de GUI uit.

De website bestaat uit de *back-end,* waar berekeningen worden uitgevoerd, en de *front-end,* wat de gebruiker ziet. De *back-end* is geschreven in Python, en gebruikt eerder besproken functies om de kortste pad en tijd te berekenen (functie *run_algorithm* (zie paragraaf 3.2) en de congestiegrafen (zie paragraaf 5.4)). De *front-end* is geschreven in HTML, met CSS voor de opmaak, en JavaScript voor de functionaliteit. De *front-end* en de *back-end* zijn gekoppeld met Flask, een hulpmiddel in Python om websites en apps te maken. De interactie tussen de verschillende bestanden is versimpeld weergegeven in figuur 30. Een pijl hierin staat voor data-uitwisseling: één bestand gebruikt de ander. De figuur weergeeft alleen de belangrijke bestanden.




*Figuur 30: uitwisseling van bestanden binnen de GUI (versimpeld)*

## 6.1 De Flask code

Flask wordt gebruikt als motor van de website. Daarom zal die code, die opgeslagen is in het bestand *app.py*, in zijn volledigheid worden verklaard in deze paragraaf. De code zal in behapbare delen worden besproken. Alle stukjes code in deze paragraaf samengenomen vormen de complete code in *app.py*. In elke alinea wordt verwezen naar de code die zich boven de alinea bevindt.

```
from flask import Flask, render_template, request, url_for, redirect
from dijkstra import run_algorithm
from datetime import datetime, timedelta
from dynamic_graphs import graph_normal, graph_for_time, add_times, subtract_times
app = Flask(__name__)
```

Deze code vormt de basis van de Flask-applicatie en importeert de benodigde modules en functies voor het navigatiesysteem. Flask wordt geïmporteerd om een webserver op te zetten en functionaliteiten zoals het inladen van HTML-pagina's (*render_template*), het verwerken van gebruikersinvoer (*request*), en het doorsturen naar pagina's (*redirect, url_for*) mogelijk te maken. De functie *run_algorithm* functie implementeert Dijkstra's algoritme om de kortste route in de school te berekenen. Daarna worden *datetime* en *timedelta* geïmporteerd om te kunnen rekenen met kloktijden. Ten slotte worden uit het bestand *dynamic_graph.py* de volgende functies geïmporteert: *graph_for_time* (zie subparagraaf 5.4.4), *add_times* (voor het optellen van twee kloktijden) en *subtract_times* (voor het verschil berekenen tussen kloktijden). Ook wordt de variabele *graph_normal* geïmporteerd, dit is de graaf van de school, ontwikkeld in hoofdstuk 2. De laatste regel code is nodig om de webserver op te starten.

```
@app.route("/", methods=["GET", "POST"])
def login():
    try:
        if request.method == "POST":
            speed_slider = round(float(request.form.get("speed_slider")) * 1.34 / 100, 2)
            srcbox_van = int(request.form.get("srcbox_van"))
            srcbox_naar = int(request.form.get("srcbox_naar"))
            time_input = request.form.get("time")
            type_tijd = request.form.get('typeTijd')
            return redirect(url_for("map", src=srcbox_van, dest=srcbox_naar, speed=speed_slider,
time_input=time_input, type_tijd=type_tijd))
        else:
            return render_template("login.html", graph=graph_normal)
    except:
        return render_template("login.html", graph=graph_normal)
```

Om een website te bezoeken is een *URL* nodig. Deze staat in de zoekbalk staat tijdens het bezoeken van de website. Een *URL* bestaat uit een domeinnaam en een *URL-path*. De domeinnaam is hier "hyperion-navigator.onrender.com". Dat staat niet geprogrammeerd in de code, maar is ingesteld op de hosting service (render.com). In de code staat de *URL-path*, dat is het gedeelte van de URL dat na de ".com" of ".nl" komt. In de bovenstaande code is de *URL-path* "/". Dat betekent dat de *URL* als volgt is: "hyperion-navigator.onrender.com/". Wanneer deze *URL* in de zoekbalk staat, wordt de bovenstaande functie *login* geactiveerd, en wordt de web-pagina *login.html* ingeladen

De inhoud van de *login* staat in een *try-block*, wat betekent dat als de code na de "try:" een error ondervindt, de code na "except:" wordt geëxecuteerd. Dit is een vangnet voor de programmeur, voor als er een kleine error het tijdelijk niet mogelijk maakt de code na "try:" te executeren.

In de *try-block* staat een *if-statement*, dat geactiveerd wordt wanneer op de *verzendknop* wordt gedrukt. De data, ingevuld door de gebruiker, worden dan omgezet in variabelen. Deze worden vervolgens doorgestuurd naar de *map* functie die hieronder wordt uitgelegd.

```
@app.route("/van-<src>/naar-<dest>/snelheid=<speed>/ingevoerde_tijd=<time_input>/type_tijd=<type_tijd>")
def map(src, dest, speed, time_input, type_tijd):
    if type_tijd == 'aankomst':
        graph = graph_for_time(subtract_times(time_input, "00:00:30"))
        route, time = run_algorithm(graph=graph, startnode=int(src), endnode=int(dest), speed=float(speed))
        time = round(time, 0)
        leave_time = (datetime.strptime(time_input, "%H:%M:%S") -
timedelta(seconds=time)).strftime("%H:%M:%S")
```

```
        arrival_time = time_input
        leave_message = 'Vertrek om'
        arrival_message = 'Als u aan wilt komen op'
    else:
        graph = graph_for_time(add_times(time_input, "00:00:30"))
        route, time = run_algorithm(graph=graph, startnode=int(src), endnode=int(dest), speed=float(speed))
        time = round(time, 0)
        leave_time = time_input
        arrival_time = (datetime.strptime(time_input, "%H:%M:%S") +
timedelta(seconds=time)).strftime("%H:%M:%S")
        leave_message = 'Vertrek om'
        arrival_message = 'Voorspelde aankomsttijd'
    time_str = f"{int(time // 60)} minuten en {round(time % 60)} seconden"
    return render_template("map.html", src=src, dest=dest, speed=speed, route=route, time=time_str,
graph=graph, leave_time=leave_time, arrival_time=arrival_time, leave_message=leave_message,
arrival_message=arrival_message)
```

De functie *map* laadt uiteindelijk het *routescherm* in, waar de gebruiker de snelste route ziet, gebaseerd op de ingevoerde gegevens.

In de functie *map* wordt allereerst de data ontvangen die vanuit de functie *login* is verstuurd. Deze data worden in de *URL-path* gezet waardoor deze de volgende structuur krijgt: "/van-<src>/naar-<dest>/snelheid=<speed>/ingevoerde_tijd=<time_input>/type_tijd=<type_tijd>". Tussen de chevrons staat de verstuurde data. Een voorbeeld van URL zou kunnen zijn: "https://hyperion-navigator.onrender.com/van-7/naar-120/snelheid=1.74/ingevoerde_tijd=14:05:10/type_tijd=vertrek". In feite worden dus de invulling in het beginscherm (de tijd, het startpunt etc.) in de *URL* geplaatst.

Daarna wordt de functie *graph_for_time* gebruikt om de juiste graaf te vinden voor het ingevulde tijdstip. Vervolgens wordt deze graaf door de functie *run_algorithm* gebruikt om het snelste pad en de looptijd van het ingevulde start- en eindpunt te vinden.

Ten slotte wordt de looptijd van het pad opgeteld bij of afgetrokken van de ingevulde tijd om een aankomsttijd of vertrektijd voor te stellen. Dit hangt af van het type tijd (vertrek of aankomst) dat gekozen is voor de ingevulde tijd.

```
if __name__ == "__main__":
    app.run()
```

Als laatste zorgen deze twee regels code ervoor dat de app wordt gestart.

## 6.2 Visualiseren graaf met JavaScript

Nadat de gebruiker informatie heeft ingevoerd in het beginscherm en op de *verzendknop* heeft gedrukt, wordt het *routescherm* ingeladen. Het visualiseren van de graaf op de plattegronden gebeurt volgens de JavaScript code in *script.js*. Deze code is grofweg voor 85% geschreven door ChatGPT. Echter, de code is niet rechtstreeks overgenomen van ChatGPT, maar is aangepast om het goed onderhoudbaar en aanpasbaar te maken. Een simpel voorbeeld hiervan is dat de kleuren van de vertices en zijden niet apart in de code worden gedefinieerd (zoals ChatGPT voorstelde), maar als globale variabelen worden gedefinieerd. Die variabelen kunnen dan meermaals in de code worden gebruikt. Op deze manier kan de programmeur makkelijker door de gehele code de kleuren veranderen.

In *script.js* wordt de kortste route (berekend door Dijkstra's algoritme) gebruikt om plattegronden in HTML te verstoppen als ze niet aanwezig zijn in de route. Als een route zich bijvoorbeeld beperkt tot de eerste verdieping, is het niet nodig om de andere verdiepingen te weergeven. Hiervoor is een lijst gemaakt in JavaScript met de z-coördinaat van alle vertices. De z-coördinaat is gelijk aan het verdiepingsnummer (kelder is -1, begane grond is 0, etc.). Zo kan worden vastgesteld welke verdiepingen zich wel en niet op de snelste route bevinden, en worden alleen de relevante verdiepingen getoond.

Voor de GUI heeft elke vertex een z-coördinaat, een x-coördinaat en een y-coördinaat, om het punt te kunnen tonen op de plattegrond. In de volgende paragraaf wordt uitgelegd hoe deze werken. *D3* (data driven documents, een JavaScript library voor het visualiseren van data) gebruikt de x en y-coördinaten van vertices, de graaf, en de route om de graaf te “tekenen” op de plattegronden van het schoolgebouw.

## 6.3 Responsive design

De intentie was een website te creëren die zowel op de computer als op de telefoon gebruiksvriendelijk is. Hiervoor is *responsive design* nodig; de indeling van een webpagina past zich aan aan de grootte van het scherm. Voor de lettertypegrootte en de afmetingen van invulvelden gebeurt dit automatisch in HTML. De grootte van witruimte om HTML elementen (tekst, invulvelden, afbeeldingen, etc.) is uitgedrukt als percentage van de breedte van het scherm van de gebruiker. Bijvoorbeeld, In de CSS staat voor de *body* (een container element waar alle zichtbare elementen zich in bevinden in HTML):

```
body {
    width: 80%;
    margin: auto;
    padding: 0px;
    text-align: left;
    border: 2px solid transparent;
}
```

De *width* (breedte) van de *body* is gelijk aan 80% van de schermbreedte. Dat betekent dat de witruimte in totaal 20% is, dus 10% aan elke kant. Wanneer de website wordt bezocht op een telefoon, zal de witruimte kleiner zijn dan op de laptop. Dit is noodzakelijk om een consistente en gebruiksvriendelijke weergave op verschillende schermformaten te waarborgen. De tegenovergestelde, non-responsive, aanpak zou zijn om de witruimte te definiëren in pixels of cm. Dat zou ervoor zorgen dat de website er heel anders uitziet op de telefoon dan op de laptop.

Een ander "trucje" van de responsive design is te werken met *max-width* en *aspect-ratio*, zoals voor de *svg* afbeeldingen (scalable vector graphics, een afbeelding die niet slechter wordt in kwaliteit als er meer wordt ingezoomd). In de CSS voor de GUI werkt het als volgt;

```
svg {
    margin: auto;
    background-size: cover;
    border: 2px solid black;
    margin-top: 5px;
    max-width: 600px;
    aspect-ratio: 11995 / 6338;
    display: none;
}
```

De *max-width* zorgt ervoor dat er een limiet is aan hoe groot de afbeeldingen kunnen worden. Op de telefoon is een zo groot mogelijke afbeelding gewenst, omdat het scherm klein is, maar op de laptop is vaak fijner voor de gebruiker als de afbeelding niet het hele scherm inneemt. *Aspect-ratio* zorgt ervoor dat de hoogte van de afbeelding automatisch wordt aangepast wanneer de breedte verandert, zodat de afbeelding niet wordt uitgerekt. Er is voor de margin-top wel voor een absolute afstand gekozen, omdat een consistente verticale afstand tussen de afbeelding gewenst is, ongeacht de schermgrootte. Dit voorkomt dat de marges te groot worden op bredere schermen en zorgt voor een evenwichtige lay-out.

Om uiteindelijk de graaf te visualiseren, worden de x- en y-coördinaten van alle vertices gebruikt. Om het design responsief te maken, is ervoor gekozen zowel de x als de y-coördinaten uit te drukken als percentage van de breedte van de afbeelding (in dit geval een plattegrond). Ook de dikte van de lijn die de snelste route aangeeft, evenals de grootte van de vertices, is uitgedrukt als percentage van de breedte van de plattegrond. Doordat deze breedte afhangt van de schermgrootte, ziet de visualisatie van de graaf er op alle typen schermen consistent uit.

## 6.4 Mappenstructuur en hosting

De website is ontwikkeld in Visual Studio Code. Deze code is geüpload naar GitHub (zie bijlage E voor de link). Vervolgens is de GitHub gekoppeld aan Render. Render is een website voor het hosten van onder andere Flask websites.

# 7 Conclusie

In dit onderzoek is een navigatiesysteem ontwikkeld dat de snelste routes vindt in het schoolgebouw van het Hyperion Lyceum en rekening kan houden met files.

Allereerst is er verdiept in grafentheorie en de mogelijkheden hiervan. Tevens is er gekeken naar Dijkstra's kortste pad algoritme, dat het kortste pad op een graaf kan vinden. Om grafentheorie te fuseren met verkeerstheorie is er verdiept in verkeersstromen op gesimplificeerde netwerken. Ten slotte is verkeerstheorie aan bod gekomen en is bestaande theorie over voetgangersstromen uitgelicht.

De volgende stap was het ontwikkelen van een graaf die de structuur van het schoolgebouw weergeven. In dit maakproces is er veel aandacht besteed aan het dilemma tussen simpliciteit en waarheidsgetrouwheid in het ontwikkelen van een graaf. Vervolgens zijn er gewichten berekend voor de zijden van de ontwikkelde graaf, resulterend in een gewogen graaf. Deze vormde de basis voor het verdere onderzoek.

Vervolgens is Dijkstra's algoritme gecodeerd in Python, waardoor de kortste paden gevonden kunnen worden.

Als opstap naar een groter verkeersmodel, is er eerst gecodeerd aan een casus van een verkeersstroom tussen twee lokalen. Het ontwikkelde model liet veelbelovende resultaten zien, dus is ervoor gekozen om de stap te maken naar een compleet stroommodel.

Het complete stroommodel simuleert de beweging van voetgangersverkeer op drukke momenten zoals tussen lessen in. Uit dit model zijn drie congestiegrafen gehaald, die gebruikt kunnen worden voor drie soorten verkeersstromen.

De graaf van het Hyperion Lyceum en Dijkstra's algoritme, samen met het selectieve gebruik van de congestiegrafen, zijn samengebracht in een website. Deze gebruiksvriendelijke interface van de ontwikkelde code kan gebruikt worden door leerlingen en bezoekers van het Hyperion Lyceum om de snelste route te vinden.

## 7.1 Hoogtepunten

Wij willen graag enkele veelbelovende en noemenswaardige aspecten van ons onderzoek uitlichten:

- Het verkeersmodel is een zelf ontwikkeld programma om voetgangersstromen te simuleren op microniveau. Elke voetganger wordt dus apart bijgehouden. Dit model kan hierdoor zeer realistische resultaten bereiken, zo blijkt uit resultaten van de casus.
- De website is een zorgvuldig ontwikkelde interface die gebruiksvriendelijk is en gemakkelijk gebruikt kan worden door onze medeleerlingen en bezoekers van het Hyperion Lyceum. Het is een samensmelting van de verschillende aspecten van ons onderzoek en een inzicht in wat wij precies ontwikkeld en bestudeerd hebben in dit onderzoek.

## 7.2 Ruimtes voor verbetering

Ook zijn er gebieden waar verbetering of verder onderzoek mogelijk is:

Een positioning system dat een locatie bepaalt op basis van de afstand tot bepaalde sensoren. Zo hoeft een gebruiker niet in te vullen waar hij/zij is. Een nieuwkomer op het Hyperion Lyceum zou bijvoorbeeld moeilijk kunnen weten waar hij/zij is op basis van de plattegronden.

Gebruik van schoolroosters in plaats van uniforme willekeur voor het bepalen welke start- en eindpunten allemaal gebruikt gaan worden. Zo kan een verkeersstroom tussen twee lessen nog accurater voorspeld worden.

Het systeem dynamischer maken door real-time informatie of gebeurtenissen mee te nemen in het bepalen van een route. Denk bijvoorbeeld aan een afgesloten gang of een dichte glijbaan. Zo zou bijvoorbeeld een interface gebouwd kunnen worden voor de conciërges om real-time informatie in te voeren. Zo kunnen foute routes voorkomen worden.

Meenemen van onzekerheden van dichte deuren. Sommige deuren (meestal tussen twee lokalen) zijn regelmatig dicht. Door de volgende verwachtingswaarde op te stellen: $E(X) = p_{dicht} t_{omlopen} + p_{open} w$. Als deze verwachtingswaarde genomen wordt als het nieuwe gewicht kan meegerekend worden dat sommige deuren vaak dicht zitten. Dit zorgt ervoor dat de gekozen routes minder door lokalen lopen, maar meer door gangen.

# 8 Discussie

## 8.1 Testen systeem

### 8.1.1 Steekproefmethodiek

Het systeem wordt empirisch getest met een steekproef. Er worden tien routes willekeurig gekozen, door de begin- en eindpunten willekeurig te laten genereren door een computer. Deze routes worden gekozen op momenten dat er geen drukte is, om te kijken of de basis van het systeem klopt. Het systeem geeft aan hoeveel tijd het kost om elke route te belopen. Dit is $t_{cor}$ (zie formule 33). Tien proefpersonen belopen de tien paden. Hun looptijd is genoteerd als $t_{echt}$, de variërende loopsnelheden worden buiten beschouwing gelaten. Om de wetenschappelijke integriteit te waarborgen vertellen de onderzoekers de proefpersonen niet wat is $t_{cor}$ is. Met deze metingen bepalen we of het systeem statistisch nauwkeurig is. Hiertoe wordt er gekeken of de gemiddelde relatieve afwijking tussen $t_{cor}$ en $t_{echt}$ statistisch te groot is. Dit is uiteraard een toevalsproces. Men identificeert dus een relatieve afwijking als stochast $X$:

$$X_i = \frac{t_{echt,\, i} - t_{cor,\, i}}{t_{cor,\, i}} \tag{47}$$

Er is aangenomen dat $X$ normaal gedistribueerd is met:

$$\mu_X = 0 \tag{48}$$ [6]

$$\sigma_X = \sqrt{\frac{1}{n-1} \cdot \sum_{i=1}^{n} (X_i - \bar{X})^2} \tag{49}$$ [7]

$\bar{X}$ is de gemiddelde relatieve afwijking van een steekproef van 10 routes ($n = 10$).

$$\bar{X} = \frac{1}{n} \cdot \sum_{i=1}^{n} X_i \tag{50}$$

Hieruit volgt dat $\bar{X}$ ook normaal gedistribueerd is met:

$$\mu_{\bar{X}} = \mu_X = 0 \tag{51}$$

$$\sigma_{\bar{X}} = \frac{\sigma_X}{\sqrt{n}} \tag{52}$$

In eerste instantie wordt aangenomen dat de gemiddelde relatieve afwijking van het systeem gelijk is aan 0. Dit wordt de nulhypothese genoemd:

$$H_0:\ \mu = 0 \tag{53}$$

[6] Volgt uit nulhypothese, zie formule 53
[7] Bessels' correctie, er zijn maar n-1 graden van vrijheid

De alternatieve hypothese (die nog niet wordt aangenomen) geeft aan dat de gemiddelde relatieve afwijking niet gelijk is aan 0.

$$H_1:\ \mu \neq 0 \tag{54}$$

Er wordt een significantieniveau van 5% genomen:

$$\alpha = 0,05 \tag{55}$$

Dit houdt in dat berekend moet worden hoe groot de kans is dat de een steekproefgemiddelde $\bar{X}$ een grotere absolute afwijking heeft dan het gemeten steekproefgemiddelde $k$, op basis van $H_0$ .

- Als deze kans kleiner is dan ½$\alpha$, wordt $H_0$ verworpen en $H_1$ aangenomen. Dit betekent dat de afwijking niet door toeval verklaard kan worden, maar een systematische fout is.
- Is de kans groter dan ½$\alpha$, dan kan de afwijking als toeval worden beschouwd, en blijft $H_0$ staan.

### 8.1.2 Uitvoer en resultaten

De resultaten van de willekeurig gekozen routes, de aangegeven tijden en de empirisch gemeten tijden, staan in de onderstaande tabel. De meest rechter kolom is de relatieve afwijking, berekend met de gegevens uit de twee middelste kolommen en formule 45.

| $i$ | $P_i$ | $t_{cor,i}$ in s | $t_{echt,i}$ in s | $X_i$ |
|---|---|---|---|---|
| 1 | (124, 125, 121, 120, 119, 118, 148) | 32,03 | 34,17 | 0,0668 |
| 2 | (37, 44, 38, 81, 80, 110, 109, 105, 104, 103, 101, 98) | 88,2 | 87.24 | -0,0109 |
| 3 | (48, 49, 50, 91, 102, 134, 135, 138, 137, 155, 156) | 96,52 | 99,52 | 0,0311 |
| 4 | (63, 66, 69, 41, 42, 43, 84, 85, 86, 88) | 39,9 | 46,61 | 0,1682 |
| 5 | (0, 1, 16, 17, 18, 24, 7, 5) | 28,96 | 33,03 | 0,1405 |
| 6 | (86, 85, 84, 78, 80, 110, 111, 143, 142, 140, 138, 137, 155, 153, 152, 151) | 71,81 | 83,06 | 0,1567 |
| 7 | (156, 155, 153, 152, 149, 148, 118, 119, 72, 71, 62, 63, 64, 65) | 75,37 | 69,17 | -0,0823 |
| 8 | (18, 24, 7, 5, 6, 35) | 15,11 | 15,20 | 0,0060 |
| 9 | (143, 111, 110, 80, 78, 84, 85, 7, 24, 18) | 64,27 | 68,57 | 0,0669 |

| 10 | (66, 69, 41, 42, 38, 81, 80, 110, 111, 143, 142, 140) | 67,61 | 69,02 | 0,0209 |
|---|---|---|---|---|

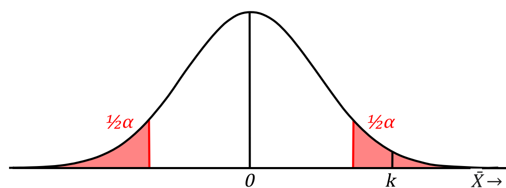
*Tabel 9: gegevens steekproef*

Uit de tabel volgt een steekproefgemiddelde $k = 0,05639$. Ook is uit de gegevens van de tabel en formule 47 te bepaald dat $\sigma_X = 0,08032$, uit formule 50 volgt dat $\sigma_{\bar{X}} = 0,02540$.

De overschrijdingskans van $k$ is:

$P(\bar{X} \geq 0,05639) = normalcdf(0,05639; 10^{99}; 0; 0,02410) = 0,01321 < 0,025$ [8]

Dus $P(\bar{X} \geq k) < ½\alpha$. Hieruit volgt dat $H_0$ verworpen moet worden en de gemiddelde relatieve afwijking groter dan 0 is. Dit betekent dat er gezegd kan worden dat het navigatiesysteem de looptijd onderschat met gemiddeld 5,6%.



*Figuur 32: normaalverdeling en steekproef*

## 8.2 Testen verkeersmodel

In hoofdstuk 5 wordt een zelfgemaakt verkeersmodel voorgesteld voor het simuleren van leerlingenstromen op drukke momenten in het schoolgebouw. De resultaten van dit model worden samen met het systeem dat de snelste paden vindt, geïmplementeerd in een applicatie. In de vorige paragraaf is de basis van dit systeem getest door middel van een steekproef. Dit model en zijn resultaten kunnen ook onder de loep genomen worden. In deze paragraaf worden enkele discussiepunten van het model en zijn resultaten aangekaart en worden er methodes voorgesteld om het model en zijn resultaten te testen.

### 8.2.1 Discussiepunten

- **Nash-gehalte van de verkeersstromen**. Aan de basis van het stroommodel staat de aanname dat leerlingen de snelste route willen nemen en geïnformeerd zijn over vertraging op de mogelijke paden. De mate waarin leerlingen geïnformeerd zijn over wat precies de snelste route is kan in twijfel getrokken worden aangezien in veel gevallen er veel mogelijke paden zijn en het zelden glashelder is welke route de snelste zal zijn. Echter, als iedereen de navigatie-applicatie gebruikt kan hier wel vanuit gegaan worden. Deze kiest immers het snelste pad.
- **Verschil in gedrag**. In de constanten, voorgesteld door Weidmann, wordt niet uitgegaan van voetgangersstromen in een schoolomgeving. Wellicht dat leerlingen meer samenklonteren en daardoor voor een verminderde doorstroom zorgen. Ook is er een evident probleem met het model, dat het niet meeneemt dat soms voor een lokaal gewacht moet worden. Voor lokaal 10, 11 en 12 is er altijd file omdat leerlingen

[8] Normalcdf is een functie op TI (Texas Instruments) grafische rekenmachines

frequent moeten wachten op het opengaan van een lokaal, wat veel file veroorzaakt. Dit is niet terug te zien in het model.
- **Samenvatting in vertices en zijden**. Allereerst zorgt een tussenliggende vertex ervoor dat het model niet in staat is om een file op één zijde over te laten slaan op een andere zijde (besproken in paragraaf 4.4). Daarnaast word er alleen gekeken naar file op zijden en niet op vertices. Er wordt dus alleen gekeken naar file op gangen, niet naar file op kruispunten.
- **Gebruik van resultaten**. De gebruikte congestiegrafen zijn genomen op momenten dat de voetgangersaantallen op een maximum lagen. Dit kan wellicht een vertrokken beeld geven van de werkelijke file, omdat er alleen gekeken wordt naar de extreme gevallen.
- **Gebrek aan empirische ondersteuning voor de invoer**. De invoer van het programma (zoals de periode waarin deuren open kunnen gaan) is grotendeels gebaseerd op eigen redenering maar heeft weinig empirische ondersteuning.

### 8.2.2 Verificatiemethoden

Een effectieve maar zeer intensieve methode tot het verifiëren van de gebruikte resultaten (de congestiegrafen) zou zijn om op verschillende dagen de looptijd op een zijde te meten, hier een gemiddelde van te nemen en deze te vergelijken met het gewicht in de congestiegraaf (bijvoorbeeld met een steekproef). Dit zou wel een groot aantal metingen vereisen, aangezien er voor drie congestiegrafen, op meerdere zijden, meerdere metingen gedaan moeten worden.
Het testen van het verkeersmodel zelf is wat gecompliceerder. Een mogelijke manier zou zijn om in te zoomen op bijvoorbeeld één kruispunt en hier het gedrag van het model te vergelijken met een werkelijke situatie. Ook zou er bijvoorbeeld een stresstest gedaan kunnen worden door grote aantallen personen door de graaf los te laten. Vervolgens kan er gekeken worden of het model logisch gedrag vertoont.

# Literatuurlijst

Buchmüller, S., & Weidmann, U. (2006). *Parameters of pedestrians, pedestrian traffic and walking facilities* (IVT Schriftenreihe 132). ETH Zürich. https://doi.org/10.3929/ethz-b-000047950

Computer Science Bytes. (z.d.) *Representing Graphs.* https://www.computersciencebytes.com/array-variables/graphs/representing-graphs/

Dijkstra, E. (1959). *A note on two problems in connexion with graphs : (Numerische Mathematik, 1(1959), p 269-271)*. Stichting Mathematisch Centrum. https://ir.cwi.nl/pub/9256/9256D.pdf

Eldridge, E. (z.d.). *Nash equilibrium.* Britannica. https://www.britannica.com/science/Nash-equilibrium

Google Maps. (2025). *Kaart Nederland.* Google.

Greenshields, B. (1935). *A study of traffic capacity.* Traffic Bureau Ohio State Highway Department. https://web.engr.oregonstate.edu/~bertinir/TFT/greenshields/docs/greenshields_1935_1.pdf

Hoogendoorn, S. P. (2016). *Traffic flow theory and simulation*. Delft University of Technology. https://ocw.tudelft.nl/courses/traffic-flow-theory-simulation/

Kladek, H. (1966). *Über die Geschwindigkeitscharakteristik auf Stadtstraßenabschnitten* (Proefschrift).

NETWORKS. (2024). *NETWORKS goes to school #3*.

Sahaleh, Sohrab & Bierlaire, Michel & Farooq, Bilal & Danalet, Antonin & Hänseler, Flurin. (2012). *Scenario Analysis of Pedestrian Flow in Public Spaces.*

Scholen op de kaart. (z.d.). *Hyperion Lyceum*. https://scholenopdekaart.nl/middelbare-scholen/amsterdam/4463/hyperion-lyceum/

Schrijver, A. (z.d.). *Grafen: Kleuren en Routeren*. Centrum Wiskunde & Informatica. https://homepages.cwi.nl/~lex/files/graphs1_3.pdf

ti84calc. (z.d.) *TI 84 calculator online.* (afbeelding). https://ti84calc.com/ti84calc

Weidmann, U. (1993). *Transporttechnik der fußgänger: transporttechnische eigenschaften des fußgängerverkehrs, literaturauswertung* (IVT Schriftenreihe 90). ETH Zürich. https://www.research-collection.ethz.ch/bitstream/handle/20.500.11850/242008/1/sr90_2auflage.pdf

Wikipedia. (z.d.). *Beta distribution pdf.* https://en.wikipedia.org/wiki/File:Beta_distribution_pdf.svg