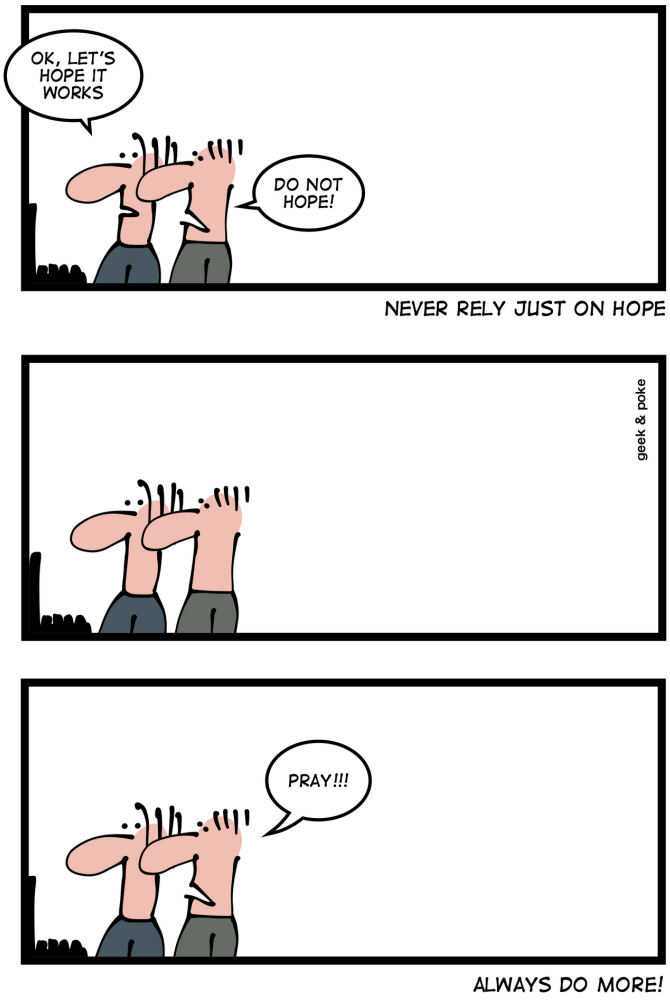
Universiteit van Amsterdam

Opgave 7

Stephen Swatman – Jordy Perlee – Robin de Vries

Simuleerverkeer



Deadline: zaterdag 21 oktober 2017, 23:00

Voorwoord

Voor je ligt de laatste opgave van Inleiding Programmeren. Echt makkelijk is hij niet, maar na de vorige zes opgaven moet het te doen zijn. We hopen dat jullie de afgelopen weken veel hebben geleerd van jullie tripje door de wondere wereld van Java programmeren en dat jullie er natuurlijk ook heel veel plezier aan hebben beleefd. Succes met de laatste opdracht!

1 Introductie

In deze opdracht maak je kennis met een van de belangrijkste subdisciplines van de informatica – het simuleren. Aan de hand van een aantal regels ga je een proces uit de echte wereld zo precies mogelijk afbeelden in een computer en je gaat aan de hand daarvan kijken naar bepaald gedrag. Daarnaast maak je kennis met algoritmen om het kortste pad te vinden tussen twee plaatsen. Het zogenaamde kortstepadprobleem komt heel vaak terug binnen de informatica; denk bijvoorbeeld aan Google Maps of aan computerspelletjes. In dit geval gaan we een van de meest bekende algoritmen ooit, het algoritme van Dijkstra (nota bene afkomstig van het CWI), toepassen om gesimuleerd verkeer goed te laten verlopen over een zogenaamde graaf. In het volgende blok ga je uitgebreid leren over grafen dus je kunt deze opdracht zien als een klein voorproefje.

De simulatie die je gaat schrijven bootst een wegennetwerk na waarop allerlei voertuigen rijden. Je gaat ervoor zorgen dat de voertuigen zich daadwerkelijk kunnen verplaatsen over het wegennetwerk en dat automobilisten bij knooppunten de optimale afslag kiezen zodat ze zo snel mogelijk bij hun bestemming aankomen.

1.1 Leerdoelen

Aan het einde van deze opdracht kun je:

- omgaan met hashtabellen,
- omgaan met grafen,
- simpele simulaties maken van problemen in de echte wereld,
- het kortste pad vinden tussen twee punten met het algoritme van Dijkstra.

2 Framework

Naast dit document hoort bij deze opdracht ook weer een framework waarin een grafische interface is geïmplementeerd. Voordat je begint met programmeren of zelfs voordat je naar de code kijkt, is het raadzaam om het programma te compileren en te draaien. Je kunt het programma uitvoeren via de `Simulation` klasse (deze neemt als argument een van de bestanden in het mapje `invoer`). Als het goed is, zie je een aantal oranje rondjes en een bewegend blauw rondje zoals in Figuur 1. Deze visualisatie ga je later vervangen door een andere visualisatie die je ook cadeau krijgt.

Daarnaast krijg je van ons een bestand `Interfaces.java` met de interfaces die je in je verschillende klassen moet implementeren. Het implementeren van deze interfaces is verplicht. Er staat wat commentaar in om je op weg te helpen met wat elke methode moet doen.

2.1 Bestanden

Het framework bestaat uit vier verschillende bestanden en een mapje vol met invoer. De inhoud van het framework is als volgt:

Figuur 1: De standaarduitvoer van het framework. De animatie komt om technische redenen niet goed over in een PDF.

`Simulation.java` In dit bestand wordt de grafische gebruikersinterface aangemaakt en wordt de wereld gemaakt.

`Vehicle.java` Hier worden de voertuigen beschreven die over de weg rijden.

`Graph.java` Hier vind je de code voor de graaf en alle algoritmen die daarmee te maken hebben.

`Interfaces.java` Dit bestand bevat alle interfaces. Er staat bij elke methode wat commentaar om uit te leggen wat je moet doen. **Pas dit bestand niet aan!**

`invoer/` De bestanden in dit mapje kun je zometeen inladen als wereld. Als je het leuk vindt mag je ook je eigen wegennetwerken maken (en dan komen die misschien volgend jaar wel in het framework).

3 De `Graph` klasse

Het netwerk bestaat, net als alle grafen, allereerst uit *vertices* (ook wel knopen, nodes) die in deze context knooppunten, kruispunten, steden of dorpen voorstellen. Uit elke knoop komt minstens één *kant* (of edge), die je kunt zien als een weg. Een kant verbindt altijd twee vertices en is symmetrisch (de graaf is daarmee een ongerichte graaf). Ook heeft een kant in dit geval een maximumsnelheid waar we later iets mee gaan doen. Als laatste is een weg natuurlijk nutteloos als er geen voertuigen op rijden. Voertuigen beginnen bij een bepaalde vertex en hebben als doel een andere vertex te bereiken. Je kunt het bijvoorbeeld zien als mensen die van hun huis naar hun werk rijden of andersom. Als een voertuig bij zijn eindbestemming is aangekomen wordt deze uit de simulatie verwijderd. Elke keer dat een auto aankomt bij een knooppunt dat niet de eindbestemming is, vraagt de auto aan het knooppunt welke kant hij het beste op kan om zijn bestemming te bereiken en volgt hij de instructies op. Dit kun je zien als de bestuurder die op de wegbewijzeringsborden kijkt.

Begin met het implementeren van de `Graph` klasse. Deze klasse bevat een lijst met kanten en een lijst

met vertices die bij een bepaalde string horen (tip: gebruik een `HashMap` om dit op te slaan). Je kunt bijna alle methoden in de `GraphInterface` nu al implementeren en we raden dit ook aan als eerste stap.

Implementeer daarna de `Vertex` en `Edge` klassen als inner class van de `Graph` klasse. Ook hier kun je bijna alle functionaliteit nu al implementeren en als dat nog niet lukt kun je altijd een stump methode achterlaten die bijvoorbeeld `null` teruggeeft. Bedenk je wat je in elk van de klassen nodig gaat hebben: een `Vertex` object heeft een bepaalde positie in de wereld en een lijst met kanten terwijl de `Edge` klasse vertices bevat voor waar hij begint en eindigt alsook een maximumsnelheid. Zorg dat de `Graph`, `Vertex` en `Edge` klassen allemaal de bijbehorende interface implementeren. De `Vertex.getNextEdge` methode mag je voor nu zo implementeren dat de knoop een willekeurige kant teruggeeft waarmee hij in verbinding staat.

4 Inlezen van de wereld

Om het opzetten van verschillende simulaties makkelijker te maken, is het de bedoeling dat je de configuratie kunt uitlezen uit een bestand. Zo'n bestand ziet er als volgt uit:

```
vertex v0 50.0 50.0
vertex v1 400.0 50.0
vertex v2 200.0 200.0

edge v0 v1 13.88
edge v1 v2 13.88
edge v2 v0 13.88

vehicle v0 v1
vehicle v1 v2
```

Het eerste gedeelte van het bestand bevat de coördinaten en naam van een vertex. Het tweede gedeelte beschrijft hoe het netwerk verbonden is. De namen zijn de namen van de nodes, waarmee wordt aangegeven dat deze twee nodes verbonden zijn en het getal daarna is de maximumsnelheid op dat gedeelte van de weg in meter per seconde. Het derde en laatste gedeelte beschrijft de voertuigen. Hier wordt aangegeven bij welke vertex een voertuig begint, en wat zijn eindbestemming is.

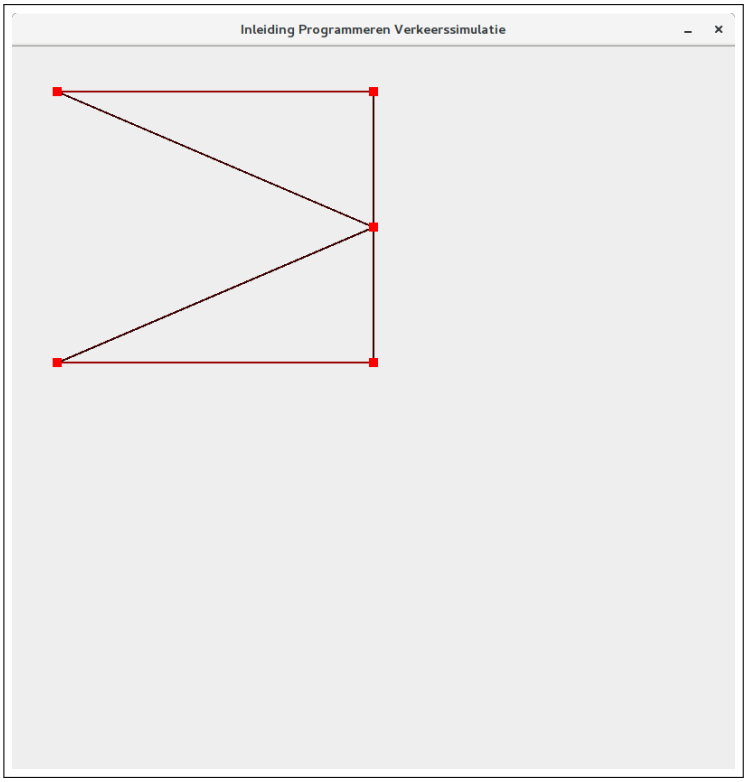
De code om deze bestanden in te lezen, is door ons gegeven in `Simulation.java:readFromFile`. In de if-statements staan nog een aantal dingen die je zelf moet implementeren. Het inlezen van voertuigen kun je nog even achterwegen laten.

Nu het mogelijk is om de configuratie uit te lezen, is het tijd om de objecten in de wereld op het scherm te tekenen. Om een simulatie beter te kunnen bestuderen, wordt er vaak gebruik gemaakt van visualisatie; het grafisch tonen van de simulatie. In de `Simulation` klasse is de visualisatie al klaargezet en uitgecomment. Uncomment in de `paint` methode de code die cadeau is gedaan voor de vertices en kanten. Als het goed is, ziet de uitvoer van het programma er daarna ongeveer zo uit als in Figuur 2. Wegen met een hogere maximumsnelheid krijgen een rodere kleur. De rode rechthoeken zijn knopen.

5 Simuleren

Op dit moment gebeurt er nog niks als je de simulatie uitvoert. De eerste stap is daarom om de voertuigen te implementeren en te laten bewegen. Behalve de `step` methode van de voertuigen zou er in de interface genoeg informatie moeten staan om de `Vehicle` klasse te implementeren. Houd daarbij rekening met de statische variabelen van die klasse: zorg dat die correct bijgehouden worden.

De Simulatie klasse zal, door middel van een timer, periodiek de `actionPerformed` methode aanroepen. Deze methode roept op zijn beurt weer de `step` methode aan waarmee stapsgewijs de simulatie uitgevoerd wordt. Deze methode moet de verstreken tijd van de simulatie ophogen, voertuigen laten verplaatsen



Figuur 2: Wat er op het scherm zou moeten verschijnen als je vertices en kanten goed kunt inlezen en tekenen. Het testbestand is `invoer/voorbeeld.txt`.

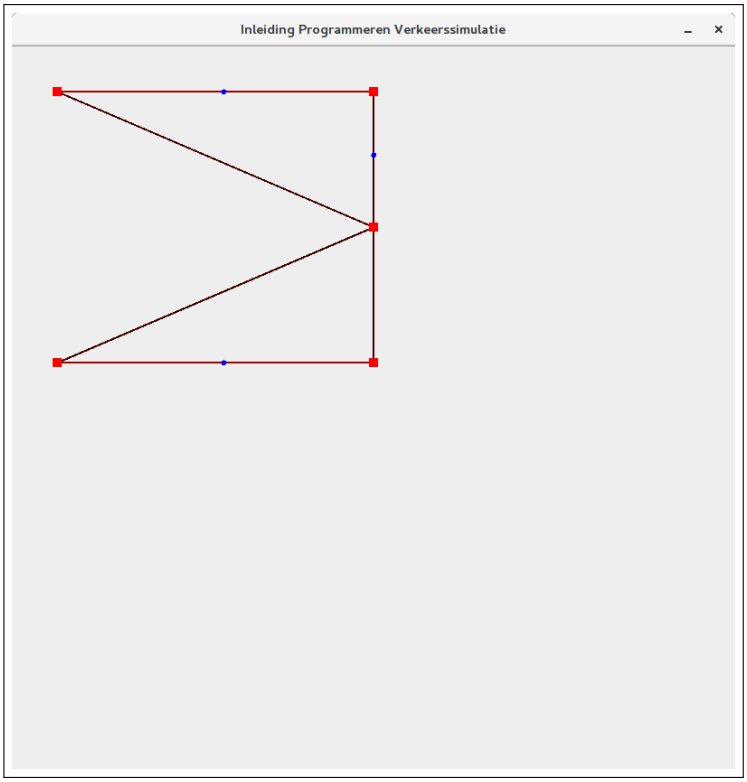
en voertuigen opruimen die bij hun bestemming zijn aangekomen. Ga voertuigen niet in de Simulation klasse bewegen maar doe dat in de Vehicle klasse. Implementeer ook de `simulationIsFinished` methode die returnt of de simulatie is afgelopen; dat is zo in het geval dat er geen voertuigen meer te simuleren zijn.

De `stap` methode van het voertuig moet twee aparte dingen doen. Ten eerste moet deze methode kijken of het voertuig bij een knooppunt is aangekomen (het kan ook zijn dat het voertuig net verschenen is en daarom nog niet op een weg rijdt) en in dat geval moet het voertuig een nieuwe richting kiezen (dit doe je door bij de knoop waar je nu bent de `Vertex.getNextEdge` methode aan te roepen en die weg te vervolgen). Ten tweede moet het voertuig een bepaalde afstand afleggen over de weg waarop deze rijdt. Hoe je dit implementeert is aan jou, maar je kunt gebruik maken van de `Simulation.speedToPixels` methode om om te rekenen tussen snelheden in meter per seconde en pixels per tijdstap.

Uncomment de code in de `paint` methode waar de voertuigen getekend worden. Als de Vehicle klasse goed is geimplementeerd bewegen de voertuigen en ziet het er op een gegeven moment uit zoals in Figuur 3 (de blauwe puntjes zijn voertuigen en kunnen elkaar overlappen). De voertuigen zouden nu bij elk knooppunt een willekeurige afslag moeten nemen.

Om te kijken hoe effectief je huidige, willekeurige 'kortstepadalgoritme' is, houd je statistieken bij over het aantal bochten dat voertuigen nemen en hoe lang ze (gemiddeld) op de weg zijn. Als je de statische variabelen in de Vehicle klasse goed bijhoudt worden deze statistieken al voor je berekend. De statistieken worden, aan het eind van de simulatie, in het volgende formaat uitgeprint:

```
Simulatietijd: 2166 (gemiddeld 1191.7)
Aantal bochten: 511 (gemiddeld 6.1)
```



Figuur 3: Wat er op het scherm zou moeten verschijnen als je voertuigen kunt simuleren. Het testbestand is wederom `invoer/voorbeeld.txt`.

6 Algoritme van Dijkstra

Om de statistieken van je simulatie te verbeteren gaan we er nu voor zorgen dat een automobilist altijd de beste weg kiest bij een kruispunt. Dat wil dus zeggen de weg die de minste reistijd geeft. Voordat je de simulatie gaat draaien gebruik je het algoritme van Dijkstra om in elk knooppunt een mapping op te zetten van vertices naar kanten waarin staat opgeslagen wat, voor elke eindbestemming, de optimale kant is om door te rijden. Gebruik hiervoor de `getWeight` methode van de `Edge` klasse. Gebruik voor het vinden van het gewicht voor elke kant de maximumsnelheid en lengte van de weg. Stel de vertices daarna zo in dat de `Vertex.getNextEdge` methode niet meer een willekeurige kant teruggeeft maar de optimale. Vergelijk de statistieken van de implementatie met het algoritme van Dijkstra met de implementatie waar automobilisten een willekeurige richting op gingen. We hebben een aantal invoerwerelden gemaakt waarmee je kunt testen of je implementatie werkt. Ben je tevreden, dan kun je de simulatie op Canvas uploaden.