DAKI Programmeeropdracht 7: Edsger op de fiets

(Deze versie is van 26 oktober 2022, 13:07)

Je moet deze programmeeropdracht **zelf** en **alleen** maken. Je mag zeker met anderen overleggen over je aanpak, maar code van anderen bekijken of overnemen of zelf code delen met anderen is uitdrukkelijk niet toegestaan. Je moet je programma schrijven in C#, en inleveren via DOMjudge.

1 Opdrachtbeschrijving

Je DAKIBOT is nu wel héél slim, en na dat puzzelen lekker ontspannen, dus besluit je dat ze een van je aankomende problemen mag gaan oplossen. Het zit namelijk zo, net als Tomas en Jelle hou jij ontzettend van fietsen. En de pandemie, waardoor je aan de DAKIBOT bent begonnen, is eindelijk bijna voorbij. Je mag zelfs voor het eerst in lange tijd weer naar de universiteit voor je hoorcolleges! Maar, omdat je zolang thuis hebt gezeten, weet je de fietsroute niet zo goed meer. Moest je nou bij dát kruispunt rechtsaf of was dat bij het kruispunt erna? Daarnaast heb jij een vervelende gewoonte om nét te lang aan je ontbijt te blijven zitten, dus heb je eigenlijk altijd haast om bij je college te komen. Daarom heb je besloten je DAKIBOT te vragen wat nou de beste (lees, snelste) fietsroute is van jouw locatie naar de universiteit. Maar als je dan toch bezig bent wil je de fietstijd van jouw locatie naar *alle* punten die je aan de DAKIBOT geeft meteen ook weten. Nu heb je ook goed op zitten letten bij De Interessante, Joyeuze, en Keurig geSTRuctureerde Academische colleges, en weet je dus al dat je dit probleem met Dijkstra's algoritme zal oplossen. (Hieronder worden termen als afstand en tijd door elkaar heen gebruikt, net als het algemenere begrip 'kosten'. De kanten hebben een *gewicht* en in die context wil je het *lichtste* (kortste, goedkoopste, snelste) pad vinden.)

Dijkstra's algoritme werkt met een priority queue waar knopen in worden opgeslagen. Omdat je die ook zelf moet implementeren moet je aan je programma een modus toevoegen waarmee we de correctheid van je implementatie van die datastructuur apart kunnen testen. Het is namelijk een goede gewoonte om afzonderlijke 'units' van je programma onafhankelijk van elkaar te testen. Als je weet dat je queue implementatie correct is, moeten eventuele problemen wel aan je implementatie van Dijkstra's algoritme liggen. (Ik raad je aan de sectie "Hints, Tips" te bestuderen.)

2 Invoer en Uitvoer

Ten eerste geldt *voor elke regel* van een testcase, net als bij eerdere opdrachten, dat de genoemde inputdata gevolgd kan worden door nog een spatie en allerlei tekst die je dan moet negeren. De eerste regel van de invoer bevat een positief geheel getal n (het aantal knopen in de graaf) en een letter t (de testmodus die je programma zal ondergaan), gescheiden door

een spatie. De modus t heeft als waarde '`p`' of '`d`' die aangeeft of de testcase je implementatie van de priority queue of van dijkstra's algoritme zal testen. Daarna volgen n regels met op elk daarvan de naam van een locatie, te weten een string zonder spaties. De volgnummers (1, 2, 3, ...) van de locaties die je inleest zijn belangrijk en moet je bijhouden en opslaan. De 'source' van het kortste pad algoritme is altijd locatie 1, dus de eerste locatie die je inleest.

modus `p`: in deze modus bevat de eerste regel na de n-de locatie een positief geheel getal x, het aantal testopdrachten. Deze opdrachten testen je priority queue implementatie. Na deze regel volgen x regels met op elk daarvan een operatie die je op je priority queue moet uitvoeren en waarvan je eventueel het resultaat moet rapporteren.

- Als een regel begint met de string '`u`', dan wordt die gevolgd door twee positieve gehele getallen i en w, beide voorafgegaan door een spatie. In dit geval moet je de prioriteit van de i-de knoop *die je hebt ingelezen* updaten naar w.
- Als een regel begint met het karakter '`e`' moet je een extract operatie uitvoeren, en de knoop met de *hoogste prioriteit* uit je priority queue halen. Je moet een regel naar de console schrijven met daarop de naam en de prioriteit van die knoop, gescheiden door een spatie.

Een extract operatie is natuurlijk pas interessant nadat er minstens één update opdracht is uitgevoerd, en de eerste van de x opdrachten is daarom altijd een '`u`' opdracht.

modus `d`: in deze modus bevat de eerste regel na de n-de locatie een positief geheel getal m, het aantal kanten in de graaf. Daarna volgen m regels met op elk daarvan drie positieve gehele getallen i, j, k, van elkaar gescheiden door spaties. Zo'n regel geeft aan dat er een kant bestaat *van* de locatie met volgnummer i *naar* de locatie met volgnummer j, met gewicht k.

In dit geval voer je Dijkstra's algoritme uit op de graaf met n knopen en m kanten. Als dat algoritme klaar is, schrijf je per bereikbare locatie één regel naar de console, met daarop de naam van de locatie gevolgd door een spatie en de kortste afstand tot die locatie vanaf locatie 1. Schrijf de locaties in de volgorde waarin je ze hebt ingelezen.

3 Voorbeeld

Bij de volgende input (een voorbeeld van de `d` testmodus):

```
7 d // Er zijn 7 locaties en je Dijkstra implementatie wordt getest
UtrechtCS         1 Het startpunt, je bent blijkbaar met het OV gekomen
Vredenburg nummer 2 nummer 2
Catharijnesingel  3 et cetera
Neude             4 en zo voort
Rubenslaan        5 en zo voort
Biltstraat        6 nummer ze6 alweer...
Universiteit      7 // nummer 7even
7 // Er zijn ook 7 kanten
1 3 2 // Van UtrechtCS naar Catharijnesingel kost 2 eenheden tijd
2 4 3 van Vredenburg naar Neude heeft kosten van 3
3 4 5 ... you get the idea
3 5 4
```

```
4 6 4
5 7 6
6 7 3
```

hoort de volgende output:

```
UtrechtCS 0
Catharijnesingel 2
Neude 7
Rubenslaan 6
Biltstraat 11
Universiteit 12
```

Bij de volgende input (een voorbeeld van de `p` testmodus):

```
7 p // Er zijn 7 locaties en je priority queue implementatie wordt getest
UtrechtCS // Het startpunt, nummer 1
Vredenburg nummer 2
Catharijnesingel et cetera
Neude       en zo voort
Rubenslaan  en zo voort
Biltstraat   nummer ze6 alweer...
Universiteit // nummer 7even
5 // Er volgen 5 operaties
u 2 5
u 3 4
e
u 6 18
e
```

hoort de volgende output:

```
Catharijnesingel 4
Vredenburg 5
```

4 Algoritmische eisen

Je moet voor deze opdracht zowel Dijkstra's algoritme als een priority queue *zelf implementeren*. Je hoeft alleen de priority queue operaties te implementeren die je voor deze opdracht nodig hebt, maar je moet de functionaliteit wel onderscheiden in de methoden zoals ze in CLRS zijn beschreven. Geef de methoden dus ook namen die duidelijk maken wat ze doen.

5 Hints, Tips

Bekijk voor priority queues hoofdstuk 6 van CLRS, en voor Dijkstra's algoritme hoofdstuk 22. Merk op dat de code van de heap algoritmen (hoofdstuk 6) een index i van de array als argument hebben, maar dat de RELAX methode (hoofdstuk 2) op een knoop u en diens buur v werkt, en eventueel de prioriteit van knoop v verandert in de heap. Om die heap-methode

(welke is dat?) efficiënt te kunnen uitvoeren, moet je *héél snel* (dat wil zeggen in $O(1)$ tijd) de index van knoop v kunnen bepalen. Daar moet je dus iets extra's voor doen.

Tenslotte moet je goed realiseren wat we hier met de 'priority' van een priority queue bedoelen. Wat probeert Dijkstra's algoritme te doen, en hoe verhoudt zich dus het gewicht van een kant (u, v) die vanuit u geRELAX't wordt (en eventueel tot een aanpassing van de waarde $v.d$ leidt) tot de prioriteit van knoop v aan de andere 'kant' van die kant? In het bijzonder: als $v.d$ lager wordt, wat betekent dit dan voor de prioriteit van v?