Programmeeropdracht 2 — Rooster
Algoritmiek, voorjaar 2021

Inleiding

Het onderwijsbureau van LIACS krijgt het steeds drukker. Met name het groeiende aantal *tracks* in de bachelor en de master levert hoofdbrekens op. Hoe maak je voor al die tracks roosters die elkaar niet in de weg zitten, en die rekening houden met de beperkingen van de docenten en de wensen van de studenten? Kun jij de medewerkers van het onderwijsbureau helpen?

Situatie

We hebben een aantal vakken n_{vakken}, een aantal docenten n_{docenten} en een aantal tracks n_{tracks}. De vakken, de docenten en tracks zijn genummerd (met aansluitende nummers), te beginnen bij 0. De vakken hebben daarnaast, voor de leesbaarheid van invoer en uitvoer, een naam. Elk vak heeft een docent, en is onderdeel van één of meer tracks. Elke docent heeft een aantal tijdsloten in de week waarop hij of zij beschikbaar is om college te geven. Elk vak moet op één tijdslot (van één uur) in de week worden ingeroosterd, in één zaal. De collegeweek bestaat uit een aantal dagen n_{dagen}, en elke dag kent een aantal uren n_{uren}. Er zijn n_{zalen} beschikbaar voor LIACS. Ook de dagen, de uren en de zalen zijn genummerd (met aansluitende nummers), te beginnen bij 0. Er geldt verder:

$$\begin{aligned} 1 \leq n_{\text{vakken}} &\leq 50 \\ 1 \leq n_{\text{docenten}} &\leq 40 \\ 1 \leq n_{\text{tracks}} &\leq 10 \\ 1 \leq n_{\text{dagen}} \times n_{\text{uren}} &\leq 50 \\ 1 \leq n_{\text{zalen}} &\leq 5 \end{aligned}$$

De een-na-laatste ongelijkheid zegt dus dat er maximaal 50 tijdsloten zijn in de collegeweek.

Doel is om, **met behulp van backtracking**, een rooster te bepalen dat aan de volgende eisen voldoet:

1. Elk vak wordt dus ingeroosterd op één tijdslot (van één uur) in de week, in één zaal.
2. Er worden geen vakken van dezelfde track op hetzelfde tijdslot ingeroosterd. Studenten kunnen dus alle vakken van hun track volgen.
3. Een docent geeft alleen college op tijdsloten waarop hij/zij beschikbaar is om college te geven.
4. Een docent geeft hoogstens één keer per dag college. Zo heeft hij/zij voldoende tijd voor de voorbereiding.
5. Een track biedt elke dag óf nul óf minstens twee vakken aan, zodat studenten niet voor één vak naar de universiteit hoeven te komen (we gaan ervanuit dat we volgend collegejaar weer fysiek onderwijs geven). Deze eis vervalt alleen voor een track als het theoretisch onmogelijk is om aan de eis te voldoen, bijvoorbeeld als een track maar één vak kent. (Bedenk zelf nog twee situaties waarin het theoretisch onmogelijk is, en verwerk die in je programma, zie ook het verslag.)

6. Een track heeft op een dag dat het vakken aanbiedt, hoogstens één tussenuur.

Voor u beschikbaar

Op de website van het vak

`https://liacs.leidenuniv.nl/~vlietrvan1/algoritmiek/`

is een skeletprogramma beschikbaar, waarin een klasse `Rooster` wordt gedefinieerd, die door het hoofdprogramma gebruikt wordt om een instantie van dit probleem in te lezen en roosters te bepalen. Het programma wordt onder Linux gecompileerd met het commando `make`. Vervolgens kun je het met het commando `./Rooster` runnen. Nadat je een instantie hebt ingelezen, kun je in een menu kiezen uit drie manieren om een rooster te bepalen.

Bedoeling is om de `TODO`'s in `rooster.cc` en `rooster.h` uit te voeren. Desgewenst kun je ook de standaardfuncties uit de bestanden `standaard.h` en `standaard.cc` weer gebruiken.

Goed om te weten: als er in het commentaar boven een functie `Pre:` staat, dan volgt de preconditie voor die functie: een aantal aannames waar je vanuit mag gaan als je in die functie binnenkomt. Je hoeft dat dus in de functie niet meer te controleren. De gebruiker van de functie moet daar van tevoren voor zorgen. Net zo staat `Post:` voor de postconditie van de functie: zaken die na afloop van de functie moeten gelden. Daar moet je in de functie dus voor zorgen.

In `rooster.h` staat gespecificeerd wat er in elke functie moet gebeuren. Voor de functies `bepaalRooster` en `bepaalMinRooster` moet er **met backtracking** een geldig rooster worden opgebouwd: een rooster dat aan alle eisen voldoet. Tenminste, als er een geldig rooster bestaat. Het verschil is dat bij `bepaalRooster` elk geldig rooster even goed is, terwijl bij `bepaalMinRooster` een geldig rooster moet worden bepaald dat 'minimaal' is, dat wil zeggen: dat zo vroeg mogelijk in de week 'klaar' is. Voor de rest van de uren en dagen in de week kan de huur voor het collegegebouw dan namelijk worden opgezegd.

Voor de functie `bepaalRoosterGretig` moet je met een gretig algoritme een rooster opbouwen. Daarbij kan het gebeuren dat je geen rooster kunt vinden dat aan alle eisen voldoet. Kies dan zelf welke eisen je het belangrijkst vindt, en probeer aan zoveel mogelijk van de eisen te voldoen.

Er is een bonus te verdienen voor de inzending(en) waarin `bepaalRoosterGretig` voor zoveel mogelijk instanties in een test een geldig (en minimaal) rooster oplevert.

Wellicht ten overvloede: zowel met backtracking als met een gretig algoritme wordt het rooster stap voor stap opgebouwd. Bij backtracking worden echter, zo nodig, stappen terug gedaan in de opbouw, om eerder gemaakte keuzes te herzien. Bij een gretig algoritme worden eerder gemaakte keuzes nooit herzien. **Als je geen backtracking gebruikt voor de functies `bepaalRooster` en `bepaalMinRooster` zal je inzending nooit voldoende zijn, hoe goed de rest van je inzending ook is.**

Een rooster dat je opbouwt, moet geretourneerd worden in een twee-dimensionaal array `rooster`, waarbij `rooster[s][z]` het nummer is van het vak dat op tijdslot s in de week, in zaal z is ingeroosterd, of -1 als er geen vak is ingeroosterd. Bij de tijdsloten nummeren we dus door: als we 9 uur per dag hebben, bestaat dag 0 uit tijdsloten 0 t/m 8, dag 1 uit tijdsloten 9 t/m 17, enzovoort. Bepaal zelf een verstandige volgorde om dit twee-dimensionale array in te vullen, als je een rooster opbouwt. Bij voorkeur een volgorde waarin je de eisen aan een geldig rooster zo vroeg mogelijk kunt controleren.

Zalensymmetrie

Als je twee vakken op hetzelfde tijdslot in verschillende zalen inroostert, maakt het voor de kwaliteit van het rooster niet uit welk vak in welke zaal geroosterd wordt. Net zo: als je twee zalen tot je beschikking hebt, en je roostert op een bepaald tijdslot maar één vak in, dan maakt het voor de kwaliteit van het rooster niet uit welke zaal je daarvoor gebruikt. Het heeft in beide gevallen dan ook geen zin om beide mogelijkheden voor het rooster te bekijken.

Verwerk deze 'zalensymmetrie' in je code voor backtracking. Dat wil zeggen: bekijk in dergelijke gevallen (ook bij meer dan twee zalen) maar een van de mogelijkheden voor de zaalindeling. Bij het gretige algoritme hoef je geen rekening met de zalensymmetrie te houden (het mag natuurlijk wel).

Invoer

- De instanties van het roosterprobleem lees je in vanuit een tekstbestand. Zo'n tekstbestand heeft het volgende, vaste formaat:
  - Een regel met drie positieve integers: n_{dagen}, n_{uren} en n_{zalen}.
  - Een regel met één positieve integer: n_{docenten}.
  - Vervolgens voor elke docent:
    - Een regel met een positieve integer $n_{\text{beschikbaar}}$: het aantal tijdsloten waarop hij/zij beschikbaar is om college te geven.
    - Een regel met $n_{\text{beschikbaar}}$ verschillende niet-negatieve integers: de tijdsloten waarop hij/zij beschikbaar is.

    De docenten zijn genummerd in hun volgorde in het bestand, te beginnen bij 0.
  - Een regel met één positieve integer: n_{vakken}
  - Vervolgens voor elk vak:
    - Een regel met een niet-lege string, bestaande uit alleen letters en cijfers (dus bijvoorbeeld geen spaties): de naam van het vak.
    - Een regel met een niet-negatieve integer en een positieve integer: het nummer van de docent van het vak, en het aantal tracks n_{tracks} waar het vak deel van uitmaakt.
    - Een regel met n_{tracks} verschillende niet-negatieve integers: de tracks waar het vak deel van uitmaakt.

    De vakken zijn genummerd in hun volgorde in het bestand, te beginnen bij 0.

  Bij het skeletprogramma zit ook een heel eenvoudig voorbeeld van een invoerbestand: `rooster1.txt`. Als het goed is, herken je daarin het hierboven beschreven formaat.

- Uiteraard betekent *in*lezen dat de uit het bestand gelezen instantie op een of andere wijze wordt opgeslagen in het betreffende object van de klasse Rooster.

  Als je in C++ een `ifstream fin` gebruikt voor je invoer-tekstbestand, dan kun je heel eenvoudig een getal of een string inlezen, bijvoorbeeld met

```
fin >> getal;
fin >> str;
```

waarbij `getal` en `str` bijvoorbeeld gedeclareerd zijn als:

```
int getal;
string str;
```

- Er zullen de komende tijd waarschijnlijk nog enkele andere .txt bestanden met instanties van het roosterprobleem beschikbaar gesteld worden.

Enkele tips bij het programmeren

- Kijk in het framework waar allemaal `TODO` staat, voor met name de functies die geïmplementeerd moeten worden.
- Het is verstandig om eerst goed na te denken over hoe je de ingelezen gegevens van een instantie op kunt slaan. Je kunt bijvoorbeeld zelf klassen `Docent` en `Vak` definiëren, voor de relevante gegevens van een docent en een vak.
- Implementeer vervolgens enkele eenvoudige functies, zoals `leesIn` en `drukAf`.

Let op: het is niet toegestaan om de headers van de gegeven public memberfuncties in `rooster.h` te veranderen. Dan zou onze automatische test (met een ander main programma) bij de beoordeling namelijk niet goed kunnen werken. Je mag wel functies toevoegen. Verder kan het inleveren van lelijke, niet-elegante, of erg inefficiënte code 0.25 punt aftrek opleveren.

Algemene opmerkingen

- Maak / behoud een verstandige klassenstructuur.
- Je klassen mogen alleen `public` membervariabelen en methoden hebben, die buiten de klasse bekend moeten zijn. Andere membervariabelen en methoden moeten `private` zijn.
- Als er in je klassen dynamisch geheugen wordt gealloceerd, denk dan ook aan een destructor.
- Functies mogen niet te lang zijn (maximaal 35 regels).
- Gebruik constantes waar dat zinvol is, zoals ook in het skeletprogramma al gebeurt.
- Het werkende programma mag er op het scherm eenvoudig uitzien, maar moet wel duidelijk zijn. De enige te gebruiken headerfiles zijn in principe iostream, iomanip, fstream, cstring, string, vector, set, unordered_set, cstdlib, climits en ctime. Wil je een andere headerfile gebruiken, vraag de docent. De headerfile algorithm mag in ieder geval niet gebruikt worden.

- Boven elke functie moet een commmentaarblokje komen met daarin een korte beschrijving van wat de functie doet. Noem daarin tevens de gebruikte parameters: geef hun betekenis en geef aan hoe ze eventueel veranderd worden door de functie. Geef bij memberfuncties ook aan wat deze met de membervariabelen van het object doen. Let verder op de layout (consequent inspringen) en op het overige commentaar bij de programmacode (zinvol en kort).
- Het programma moet onder Linux bij LIACS getest zijn en werken. Dat kan bijvoorbeeld op de huisuil, zie de instructie op de website.

Verslag

Het verslag moet getypt zijn in LaTeX, en moet bevatten:

- Een korte introductie, met uitleg over de eisen aan een rooster en uitleg over de opdracht.
- Een paragraaf waarin je uitlegt hoe je met behulp van backtracking een rooster opbouwt. In welke volgorde vul je het rooster in? Welke extra datastructuren gebruik je om de eisen aan een rooster efficient te kunnen controleren?
- Een paragraaf met nog twee voorbeelden van situaties waarin het theoretisch niet mogelijk is om een rooster te bepalen waarin elke track elke dag óf nul óf minstens twee vakken aanbiedt.

  Op pagina 1 noemen we al één zo'n situatie. In je programma moet je ook rekening houden met deze (totaal drie) situaties.
- Een paragraaf waarin je uitlegt hoe je bij het opbouwen van roosters rekening houdt met de zalensymmetrie.
- Een paragraaf waarin je uitlegt hoe je op gretige wijze een rooster opbouwt. In welke volgorde houd je rekening met welke eisen? Welke eisen zijn voor jou het belangrijkst?
- Een paragraaf waarin je een voorbeeld beschrijft waarvoor je gretige algoritme
  - geen rooster oplevert dat aan alle eisen voldoet, terwijl er wel zo'n rooster bestaat,
  - of een rooster oplevert dat weliswaar aan alle eisen voldoet, maar dat niet minimaal is.

  Dit moet een eigen voorbeeld zijn, dus geen voorbeeld dat bij de opdracht is beschikbaar gesteld. Lever ook een concreet .txt bestand met je voorbeeld in, dat door de functie `leesIn` kan worden ingelezen.
- Een 'appendix' met je complete programma (alle .cc/.h bestanden).

Aanvullingen / tips / vragen

Eventuele verdere aanvullingen of tips bij de programmeeropdracht komen op de website van het vak

`https://liacs.leidenuniv.nl/~vlietrvan1/algoritmiek/`

Daar is ook een subpagina met onder andere een template voor het LaTeX-verslag. De behaalde cijfers komen te zijner tijd in Brightspace te staan.

Heb je vragen over de opdracht, dan kun je die uiteraard stellen tijdens de practicumbijeenkomsten en de vragenuren van het vak. Je kunt ook emailen naar algoritmiek@liacs.leidenuniv.nl

In te leveren

Vorm in Brightspace een groepje voor deze opdracht, en upload:

- je programma (alle .cc/.h bestanden en Makefile voor Linux bij LIACS),
- een eigen tekstbestand met een instantie van het roosterprobleem (zie bij het verslag),
- en een PDF van je verslag (inclusief het programma!)

samen in één .zip, .tgz of .tar.gz bestand. Vermeld overal duidelijk de namen van de makers. Lever geen object-bestanden (.o) of executables in.

Uiterste (!) inleverdatum:
Voor studenten Informatica: vrijdag 30 april 2021, 23.59 uur.
Voor studenten Kunstmatige Intelligentie: vrijdag 7 mei 2021, 23.59 uur.
Teams van studenten uit beide richtingen mogen kiezen. Maak je gebruik van de latere deadline, vermeld dan expliciet welke student(en) in je team de richting Kunstmatige Intelligentie volgt/volgen.

Normering: werking 5.5 punt; commentaar en layout 1 punt; modulaire opbouw en OOP 1 punt; verslag 2.5 punt; bonus voor 'beste' gretig algoritme 0.25 punt.