Obligatorisk oppgave 5: Labyrint

INF1010

Frist: mandag 24. april 2017 kl. 12:00

Versjon 1.0 (`1709ba6`)

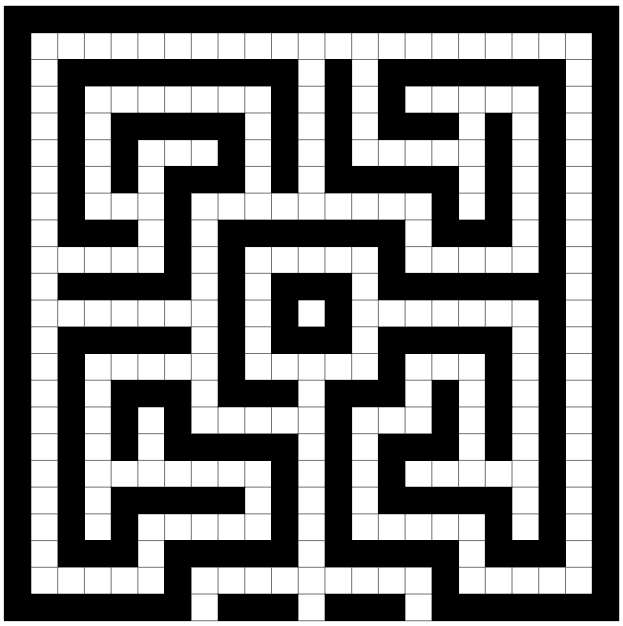
Innhold

1 Innledning

I denne oppgaven skal du bruke rekursjon til å lage et program som er i stand til å finne veien ut av en labyrint. Labyrintene i denne oppgaven er bygd opp av kvadratiske ruter som man enten kan gå gjennom eller ikke (à la labyrinter laget av hekker).

I oblig 7 skal du lage et *grafisk brukergrensesnitt* (eng. *graphical user interface*, ofte forkortet til GUI) til programmet du lager i denne oppgaven.

Figur 1: Labyrint laget av hekker (Foto: Karen Blaha på Flickr)



Figur 2: Labyrinten fra figur 1 (fil 4). Merk at labyrinten er syklisk.

2 Notasjon og terminologi

Under følger to tekster som definerer en del begreper vi får bruk for senere. Den første er strukturert som en serie presise og minimale definisjoner, som i matematikk, og den andre er strukturert som en mer uformell, forklarende tekst. Det er ikke nødvendig å lese begge.

2.1 Formelle definisjoner

- En *rute* er den minste enheten i labyrinten (hver rute er representert ved ett tegn i filen).
  - *Hvite ruter* **kan man** gå gjennom
  - *Sorte ruter* **kan man ikke** gå gjennom
- To ruter er *naboer* dersom de har en felles side.
- En *vei* er en sekvens av hvite ruter slik at hver rute i sekvensen er nabo med den foregående og den etterfølgende ruten, og disse to kan ikke være samme rute.
- Hvis det går en vei mellom to ruter, så er rutene *tilkoblet* hverandre.

- En vei er *syklisk* hvis minst én av rutene på veien besøkes flere ganger, ellers er den *asyklisk*.
- En labyrint er *syklisk* hvis det finnes en syklisk vei i den, ellers er den *asyklisk*.
- En *åpning* er en hvit rute på kanten av labyrinten.
- En *utvei* fra en bestemt rute er en vei fra den valgte startruten til en åpning.
- (x, y) betyr ruten på kolonne x og rad y. (Vi begynner å telle fra 1 og oppover).

2.2 Mindre formelle definisjoner

En labyrint er bygd opp av kvadratiske ruter som alle er like store. Hver rute svarer til et tegn i filen. En rute kan være *hvit*, som betyr at man kan gå gjennom den, eller *sort*, som betyr at man ikke kan gå gjennom den. Vi sier at to ruter er *naboer* hvis de har en felles side. En rute har altså inntil 4 naboer.

Vi kan gå mellom et par av hvite ruter som er naboer. Vi kan sette sammen mange slike veistykker for å danne en *vei*. Hvis det går en vei mellom to ruter, så er rutene *tilkoblet* hverandre.

Vi sier at en vei er *syklisk* hvis en av rutene som ligger på veien besøkes flere ganger, og en vei som ikke er syklisk kalles *asyklisk*.

Vi sier at en labyrint er *syklisk* dersom det finnes en syklisk vei i den. (Dette innebærer at vi må gjøre noen ekstra sjekker i koden for å unngå å bli gående i en sirkel). Figur 2 inneholder blant annet en syklisk vei rundt midten, (10, 10) – (14, 10) – (14, 14) – (10, 14) – (10, 10). En labyrint som ikke er syklisk, kaller vi *asyklisk*.

Hvite ruter på kanten av labyrinten kalles *åpninger*, og en vei fra en gitt rute til en åpning kalles en *utvei*. Vi kan referere til en bestemt rute ved å bruke koordinatene, altså betyr (x, y) ruten på kolonne x og rad y (og vi teller fra 1 og oppover). F.eks. har figur 3 to åpninger; (1, 2) og (13, 12).

3 Noen forenklende antagelser og nyttige resultater

3.1 Forenklende antagelser

- Vi skal i denne oppgaven bare se på rektangulære labyrinter. (Altså er den ytre grensen et rektangel.)

- Vi skal anta at ingen åpninger er nabo med hverandre. Når vi har funnet en åpning, trenger vi altså ikke å sjekke om det finnes veier videre derfra. Og dersom vi starter på en åpning, så har vi funnet en utvei og trenger ikke å lete videre.

Merk.

Vi kan ha områder med hvite ruter som er isolert fra de andre hvite rutene, som f.eks. midten i figur 2 eller området nord-øst i figur 3.

3.2 Nyttige resultater

Når du løser oppgaven er det greit å ha kjennskap til en del resultater, slik at du kan gjøre riktige antagelser når du skriver programmet ditt. Noen av disse er ganske åpenbare, men ha dem likevel i bakhodet senere.

- Dersom det ikke finnes noen sykliske veier fra en rute, så finnes det kun én vei til alle tilkoblede åpninger, og antall utveier fra ruten er dermed lik antall åpninger som er tilkoblet ruten.
- Dersom det går en syklisk vei mellom to ruter, går det også en asyklisk vei mellom dem. Det er med andre ord aldri nødvendig å bruke en syklisk vei for å nå en bestemt rute, og **vi skal derfor bare finne asykliske veier**.

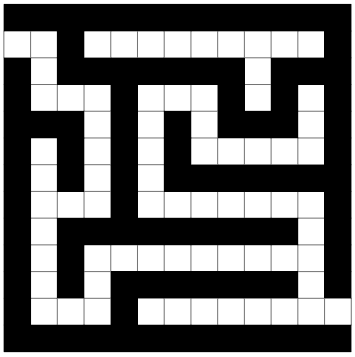
4 Filformat

For å representere labyrinter bruker vi følgende filformat: (dette er fil 3 i test-filmappen)

```
13 13
#############
..#.........#
#.#######.###
#...#...#.#.#
###.#.#.###.#
#.#.#.#.....#
#.#.#.#######
#...#.......#
#.#########.#
#.#.........#
#.#.#######.#
#...#........
#############
```

- Den første linjen inneholder to positive heltall som bestemmer hhv. antall rader og antall kolonner i labyrinten.
- Hvite ruter representeres ved `.` (punktum)

- Sorte ruter representeres ved `#` (hash/skigard)



Figur 3: Labyrinten i fil 3.

Du finner flere filer på `http://folk.uio.no/inf1010/v17/oblig/5/samples/`

5 Klasser og datastruktur

Under skisséres klassehierarkiet og datastrukturen som skal brukes. Du skal lage klassene `Labyrint`, `Rute`, `HvitRute`, `SortRute` og `Aapning`. Du står fritt til å utvide denne løsningen slik du vil, men da må du begrunne dette gjennom kommentarer i koden.

5.1 `Labyrint`

`Labyrint` har et 2-dimensjonalt `Rute`-array med referanser til alle rutene i labyrinten og instansvariable som lagrer antall rader og kolonner i labyrinten.

I tillegg skal metoden `String toString()` fra `Object` overstyres (override) slik at labyrinten kan skrives ut i terminalen. (Dette er nyttig for debugging av programmet underveis.) Du kan bruke den samme representasjonen som i filformatet, men det kan være greit å representere hvite ruter ved et mellomrom istedenfor et punktum.

5.2 `Rute`

`Rute` skal være abstrakt. Klassen skal ha instansvariable som lagrer koordinatene (rad og kolonne), samt en referanse til labyrinten den er en del av. I tillegg skal

klassen ha instansvariable for nord/syd/vest/øst som skal peke på eventuelle naboruter i de retningene.

`Rute` skal ha en abstakt metode `char tilTegn()` som returnerer rutens tegnrepresentasjon i en fil som beskrevet i 4.

5.3 `HvitRute` og `SortRute`

`HvitRute` og `SortRute` er subklasser av `Rute`. Disse må implementere `char tilTegn()` som deklareres i `Rute`.

5.4 `Aapning`

`Aapning` er en subklasse av `HvitRute` og bruker samme datastruktur som sin superklasse.

5.5 Oppsummering

- En labyrint skal representeres ved et objekt av `Labyrint` som har referanser til alle rutene gjennom et 2-dimensjonalt array.
- `Rute` er en abstakt klasse som har instansvariable
  - med koordinatene til ruten
  - med referanser til `Rute` i retning nord/syd/vest/oest
- `HvitRute` og `SortRute` er subklasser av `Rute`
- Åpninger i labyrinten er objekter av `Aapning`, som er en subklasse av `HvitRute`

6 Innlesning

Lag klassene `Labyrint`, `Rute`, `HvitRute`, `SortRute` og `Aapning` som beskrevet over.

Skriv den statiske metoden `Labyrint lesFraFil(File fil)` i `Labyrint` som først leser inn labyrinten i filen, og så kaller en `private` konstruktør i `Labyrint`, før den til slutt returnerer det resulterende objektet[1]. Det returnerte objektet skal ha all datastruktur ferdig oppsatt. Du kan anta at input-filen er gyldig.

[1] Dette er et programmeringsmønster som kalles static factory method. For å opprette nye Labyrint-objekter utenfra, må man kalle `lesFraFil(File fil)` (siden den eneste konstruktøren er `private`). Dette er en form for innkapsling som skal garantere at datastrukturen i `Labyrint` alltid blir satt opp riktig (fordi konstruktøren bare kan kalles fra metoder i `Labyrint`).

Denne metoden skal ikke håndtere `FileNotFoundException`, men bare kaste unntaket videre. Da flyttes ansvaret for å håndtere dette unntaket opp til metoden som kaller `lesFraFil`. Dette blir viktig når du skal lage et GUI til dette programmet i oblig 7.

Hint. Det kan være lurt å ha en egen hjelpemetode som avgjør om en rute er en åpning eller ikke.

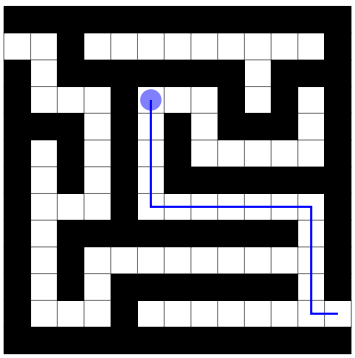
Test at brettet er lest inn riktig ved å skrive det ut igjen i terminalen.

7 Løsning ved rekursjon

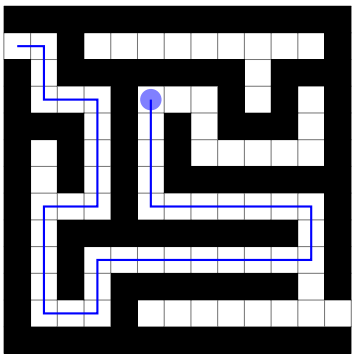
Du skal bruke rekursjon for å finne eventuelle utveier fra en bestemt rute. Idéen bak er å spørre alle naboruter om å finne en vei videre. Naboruten spør så igjen alle sine naboruter (unntatt tilbake til der kallet kom fra) om å finne en vei videre, osv. På denne måten vil vi til slutt komme til en åpning (hvis det finnes en slik).

Strategien vi skal bruke har en analogi i virkeligheten: Hvis man alltid holder til venstre (altså følger veggen som er på venstre side i forhold til retningen man går), sørger man for at man går alle de mulige veiene. I programmet vårt er det ikke nødvendig å faktisk holde til venstre, så lenge vi passer på at vi prøver å gå til alle veiene utenom den vi kom fra.

Programmet vårt skal finne alle utveier fra en rute i en asyklisk (ikke-syklisk) labyrint. Som en valgfri ekstraoppgave, kan du utvide denne løsningen slik at programmet også kan håndtere sykliske labyrinter.



Figur 4: Den første av to utveier fra (6, 4) i labyrinten i fil 3 (figur 3)



Figur 5: Den andre av to utveier fra (6, 4) i labyrinten i fil 3 (figur 3)

7.1 Utveier i asykliske labyrinter

Lag en metode `gaa` i `Rute`. Denne metoden skal kalle `gaa` på alle naboruter unntatt den som er i den retningen kallet kom fra (for da ville vi gått tilbake til der vi nettopp var).

Tips. Det kan være lurt å lage en hjelpemetode for hver av retningene man kan komme fra.

Hint. Kan du utnytte at rutene er objekter av forskjellige klasser?

Tips. Det kan være lurt i utviklingsfasen være lurt å skrive ut rutens koordinater ved hvert kall til `gaa` for å se hva som skjer.

Lag så metoden `void finnUtvei()` i `Rute` som finner alle utveier fra ruten ved å kalle `gaa`.

Lag til slutt metoden `void finnUtveiFra(int kol, int rad)` i `Labyrint` som finner utveien fra `(kol, rad)` (husk at vi begynner å telle på 1, jf. seksjon 2).

Du kan teste programmet ved å lese inn fil 3 (vist i figur 3) og kalle `finnUtveiFra(6, 4)`. De to åpningene er (1, 2) og (13, 12) og begge kan nås fra (6, 4). Utveiene er tegnet opp i figur 4 og 5. Sjekk så at det ikke finnes noen utveier fra (6, 2).

7.2 Lagring av utveien i en `String`

Endre `gaa` slik at den tar inn en ekstra `String`-parameter som inneholder koordinatene til rutene på den foreløpige veien. Når du kommer til en ny rute, legger du denne rutens koordinater til på slutten av strengen før du kaller den rekursive metoden i nabo-rutene.

Merk.

Vanligvis når vi sender med en referanse som parameter til en metode, så kan metoden forandre det objektet som parameteren refererer til. Men dette gjelder ikke objekter av klassen `String`, siden disse er *ikke-foranderlige* (eng. *immutable*). Når et `Rute`-objekt legger til sine koordinater på slutten av strengen, lager den i realiteten et helt nytt `String`-objekt.

La strengen være på formen `(x, y) -> (x, y) -> (x, y)`. (Se eksempler i 7.3

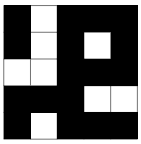
Når du har funnet en utvei, skal strengen med utveien legges inn i en `Liste<String>` (du kan selv velge hvilken av subklassene fra oblig 3 du vil bruke). Endre metoden `finnUtveiFra(int kol, int rad)` i `Labyrint` til å returnere denne listen. Husk at det skal være mulig å kalle denne metoden flere ganger på forskjellige koordinater.

Hint. Du kan lagre listen med utveier som en instansvariabel i `Labyrint`.

7.3 Hovedprogram

Det er gitt et hovedprogram på https://github.uio.no/inf1010/oblig-prekode_v17. Dette programmet leser inn filnavnet til en labyrint fra kommandolinjeargumentene og oppretter et `Labyrint`-objekt. Deretter åpner programmet en `Scanner` på `System.in` og leser inn én og én linje med koordinatene til én rute og skriver ut utveiene fra den ruten.

Dette gjør at man enkelt kan teste utveier fra mange ruter i samme labyrint. Vi bruker en veldig liten labyrint for å illustrere hvordan programmet virker.



Figur 6: Labyrinten i fil 7.

Se kjøreeksempel: (Husk at koordinatene er med kolonnen først, så raden.)

```
$ java Oblig5 7.in

2 2
(2, 2) --> (2, 3) --> (1, 3)
(2, 2) --> (2, 1)

4 2
Ingen utveier.
```

```
4 4
(4, 4) --> (5, 4)
```

Det kan være at ditt program gir en annen rekkefølge på utveiene fra én bestemt rute, men det gjør ikke noe. Det vesentlige er at det finner de samme utveiene.

OBS:

Hovedprogrammet vil kalle `settMinimalUtskrift()` på labyrint-objektet. Du må sørge for at dette gjør at programmet ikke skriver ut noe annet enn eventuelle feilmeldinger. Det kan være lurt å ha ekstra utskrifter ellers i programmet, men det må altså kunne slås av.

7.4 Oppsummering

- Programmet skal ved hjelp av rekursjon klare å finne alle mulige utveier fra en vilkårlig rute.
- Vi skal bare finne asykliske veier.
- For hver utvei vi finner, skal vi lagre en `String` som beskriver utveien. Disse strengene skal returneres i en `Liste<String>`.
- Det skal være mulig å skru av all utskrift utenom feilmeldinger.
- **Valgfritt:** (se neste seksjon)
  - Underveis skal instansvariabelen `paaVeien` i Rute oppdateres for å vise om den er allerede er på den nåværende veien, og på denne måten skal programmet også kunne finne alle utveier i sykliske labyrinter.
  - Utveiene skal lagres i en beholder slik at vi etter å ha funnet alle, kan vise den korteste.

---

8 Valgfri del

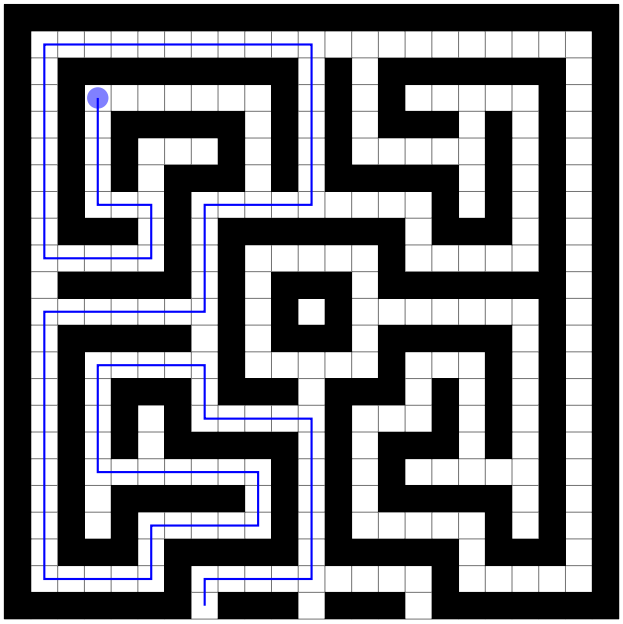
Merk.

Resten av oppgaven er valgfri, men anbefalt å gjøre.

8.1 Utveier i sykliske labyrinter (valgfri)

For å kunne håndtere sykler, må vi tilføre algoritmen vår et nytt element – vi trenger å merke veien vi har gått slik at vi kan snu når vi kommer til en rute som allerede er på veien. Vi kan se for oss at vi ruller ut en snor etter oss, og når vi går tilbake ruller vi opp igjen snoren slik at den kun ligger innenfor rutene som ligger på veien. Denne teknikken kalles *rekursiv tilbakesporing* (eng. *recursive backtracking*).

Det er greit å ha i bakhodet at vi skal finne *enhver* utvei. I sykliske labyrinter vil hver sykel kunne gi opphav til mange utveier. Noen av disse er åpenbart suboptimale, men det betyr ikke at de ikke er korrekte. Se på figur 7 og 8. Utveien i figur 7 er åpenbart fryktelig ineffektiv – faktisk er den 3 ganger så lang som den i figur 8!

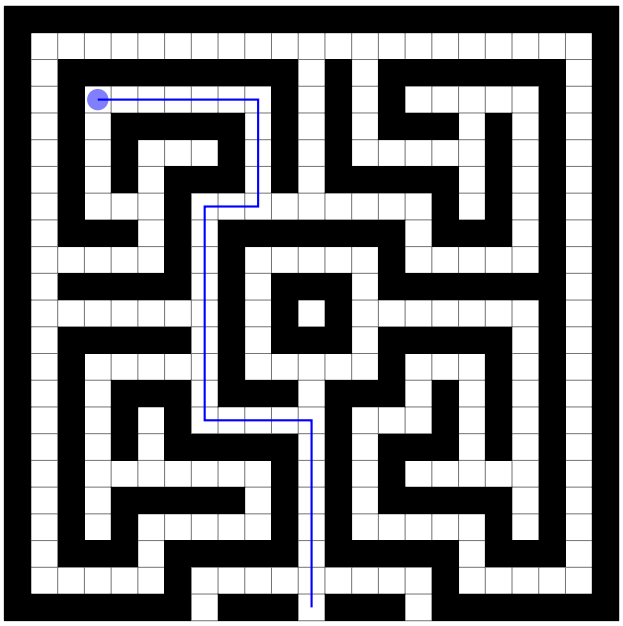


Figur 7: En tilfeldig utvei fra (4, 4) i labyrinten i fil 4 (figur 2)

Utfordring.

Klarer du å se hvor mange utveier som finnes fra (4, 4) i figur 2?

Utvid `Rute` med en instansvariabel `boolean paaVeien` og endre `gaa` slik at denne oppdateres fortløpende og hele labyrinten skrives ut når en utvei blir



Figur 8: Den korteste utveien fra (4, 4) i labyrinten i fil 4 (figur 2)

funnet. Utvid også `tilTegn()` slik den returnerer `*` (asterisk) hvis ruten ligger på veien.

Hvis du endrer programmet til å skrive ut labyrinten ved hvert tall til `gaa`, vil du kunne se hvordan algoritmen stegvis leter seg frem. (Dette blir mye utskrift, men kan være nyttig til debugging.)

Test programmet på nytt og sjekk at utveien vises riktig.

Bruk informasjonen som nå ligger i `Rute` og endre `gaa` slik at kallkjeden stopper når den kommer til en rute som allerede ligger på veien.

8.2 Korteste utvei (valgfri)

Utvid programmet ditt slik at det finner den (eller én av de) korteste utveien(e) (hvis det finnes noen utveier). Skriv også ut hvor mange utveier som ble funnet.

Hint. Det er lurt å lage en klasse for utveiene som implementerer `Comparable` slik at de kan sorteres etter lengde. Du kan bruke `OrdnetLenkeliste` fra oblig 3 til å lagre utveiene.

Lykke til!
Stein Gjessing, Stein Michael og Kristian