Varieté

Oblig.3

Contents

1 Oppgaven

går ut på å benytte en gitt modul for grafisk grensesnitt til å visualisere kjøring av noen tidligere programmerte og noen nye algoritmer.

1. Du får gitt en modul `gridFrame` som er en enkel matrise-basert grafisk grensesnitt.

Du skal så

2. tillempe noen tidligere løsninger slik at de kan brukes her som egne moduler
3. programmere noen nye algoritmer
4. og så binde det hele sammen slik at
   (a) alle programmer fra 2 of 3 kan "plugges inn"
   (b) og knyttes til det grafiske grensesnittet `gridFrame`, slik at kjøring kan visualiseres

Det er essensielt at programmet ditt skal kunne håndtere flere andre algoritmer dersom man senere får lyst til det. Det må mao. benytte en passende struktur der de forskjellige algoritmer aksesseres gjennom et felles grensesnitt.

2 Problembeskrivelse

2.1 `gridFrame.java`

Spesifikasjon og nærmere beskrivelse av denne klassen følger i seksjon 6.1 så her nøyer vi oss med å si at den gir et passende grensesnitt for sluttbruker (som kan velge mellom forskjellige display-modi, spesielt om brikkene skal vises som tall eller som fargete sirkler). For programerer, gir den hovedsakelig et grensesnitt som tillater en å plassere/fjerne brikker på spesifiserte felt, forandre dimensjoner av matrisen, mm. Grensesnittet skal være tilstrekkelig for oppgavens formål og vi antar at du bruker et objekt `gf` av denne typen, dvs. at du sier et eller annet sted i programmet ditt `gridFrame gf = new gridFrame(...)`

2.2 Krav til løsning

Programmet skal, som sagt, brukes for å visualisere kjøring av noen algoritmer på en matrise. Mht. matrisen skal bruker kunne

1. bestemme matrise-dimensjoner
2. føre inn/fjerne elementer i/fra matrisen (fortrinsvis, gjennom `gf` som musklikk)
3. få generert tilfeldig innhold i matrisen (randomBoard)
4. nullstille matrisen og hele tilstanden (starte på nytt)
5. lagre/hente matrisen til/fra en fil

Alle disse operasjonene skal reflekteres både på display'et (`gf`) og i algoritmene (som får en ny matrise-størrelse, nye brikker i matrisen, osv.) Mht. til hver enkel algoritme, skal brukker kunne

6. utføre et steg av algoritmen (hva et steg er vil avhenge av algoritmen; for noen algoritmer har vi angitt forventet betydning av et steg; for andre får du selv bestemme hvordan det defineres men der krever vi at et steg ikke resulterer i en fullstendig utførelse av hele algoritme; samt at etterfølgende steg skal, tilsammen, føre til at hele algoritme blir utført.)

7. utføre et antall (oppgit av bruker) steg av algoritmen

8. gå tilbake (minst) ett steg

Endelig, skal programmet ha en liste med forskjellige algoritmer slik at man kan

9. veksle mellom forskjellige algoritmer

Dette betyr spesielt at etter å ha kjørt en algoritme noen steg, kan man gå over til en annen algoritme uten å forandre på selve tilstanden i matrisen: man kan fortsette kjøring av en ny algoritme med den samme tilstanden/matrisen. (Dette forutsetter at de to algoritmene er 'kompatible', f.eks. hvis den første arbeider med en vilkårlig matrise mens den andre krever en matrise med lik antall rader og kolonner, og den aktuelle matrisen har ulik antall av begge, må programmet foreta en passende aksjon. Det at matrise i en algoritme representerer 'en by' mens i en annen 'en graf' er uvesentlig her.

Vi har også unntakk fra dette kravet for algoritmer beskrevet i 3.2.)

3 Gamle algoritmer

Du skal hente noen algoritmer som du programmerte tidligere i kurset og tillempe de slik at de kan brukes her.

3.1

Du skal ta med følgende algoritmer:

1. Generisk MergeSort (stigende og avtakende)

2. Transitiv tillukking av en rettet graf

3. Korteste stier mellom alle par av noder i en vektet graf

Merk at visualisering av algoritmene slik den er beskrevet over krever egentlig en iterativ implementasjon: å utføre ett steg tilsvarer å gå gjennom løkken en gang. Det er vanskelig (om i det hele tatt mulig) å gjøre noe tilsvarende for et rekursiv program, siden man har ikke noen midler til å "fryse" tilstanden i rekursjonstre for å gjenoppta eksekvering fra samme tilstand i neste steg.

De to siste algoritmene er iterative. For MergeSort har du laget en iterativ implementasjon av generisk MergeSort på øvelser. Her skal vi sortere tall, så du vil bruke en spesialisering av denne. Dersom din implementasjon ikke var "in-place", må du her finne en måte å vise stegene av algoritmen – det kan være greitt å rett og slett kopiere, i hvert steg, innholdet av interne data (sekvenser som algoritmen jobber med i øyeblikket) til den tabellen som vises for brukeren.

3.2 Dronning og SkyLine

I tillegg skal du hente løsningene til to tidligere obligatoriske oppgaver. Disse var rekursive og det kan være litt problematisk (dog langt fra umulig) å visualisere de på samme måte som algoritmene over. Derfor forenkler vi kravene her slik at visualisering av

4. Dronning-oppgaven viser kun løsningene (ett *Step* viser neste løsning). Her kan man rett og slett kjøre algoritmen og samle alle løsningene, som så vises etterpå for hvert kall til *Step*. Legg også merke til at denne algoritmen ikke bryr seg om input fra brukeren utover det at han skal kunne oppgi størrelsen på matrisen.

5. SkyLine viser kun resultatet av eliminering av skjulte linjer. Her krever vi ikke at bruker "tegner" input by på matrisen. Når denne algoritmen er aktiv, skal bruker kunne laste inn en by fra en fil (formatet fra oblig.2 skal brukes) som så vises på matrisen (f.eks. etter første kall til *Step*). Deretter (neste kall til *Step*) beregner algoritmen SkyLine og viser denne.

Har du lyst og tid, kan du selvfølgelig forsøke å gjøre denne delen mer spennende, men det er ikke noen krav.

4 Nye algoritmer

Besvarelsen skal inneholde programmer for følgende to problemer.

4.1 Største region

Vi har gitt et to-dimensjonalt bilde (en matrise) der hvert felt har en av to verdier (la oss si, 0 eller 1) – på figuren tilsvarer svarte prikker verdi 1. En *sammenhengende region* består av svarte felt der det er mulig å gå fra ethvert svart felt, via kun svarte naboer, til ethvert annet svart felt i regionen. Naboer til et felt er de vertikale og horisontale naboer – hvert felt har fire naboer: en over, en under, en til venstre og en til høyre, dvs. det midterste feltet $x : y$ har bare naboer som er merket under

	$\_ : -1$	
$-1 : \_$	$x : y$	$+1 : \_$
	$\_ : +1$	

Arealet av en region defineres som antall svarte felt som den består av.

			•	
		•	•	•
	•	•		•
•	•			•
	•			
	•	•	•	

a)

			•	
		•	•	•
	•			•
•				•
	•	•	•	
	•	•	•	

b)

			•	•
•				•
•	•			•
				•
	•	•	•	
	•	•	•	

c)

I figur a) har vi en stor region, i figur b), har vi 4 regioner (2 av dem med bare ett felt) og i figur c) har vi 3 regioner.

Skriv en algoritme som, gitt en slik matrise finner største sammenhengende region og beregner dens areal. Visualisering skal indikere, i hvert steg, en region som behandles i øyeblikket samt, til slutt, hvilken region som er den største.

4.2 Conway's 'Life'

Spillet (oppfunnet av engelsk matematiker John Horton Conway) foregår på et vilkårlig stort, rektangulært brett der det i utgangspunktet plasseres vilkårlig noen brikker. Brikker kan overleve, dø eller bli født fra genereasjon til generasjon etter følgende regler som anvendes samtidig for alle brikkene i en aktuell generasjon. En brikke

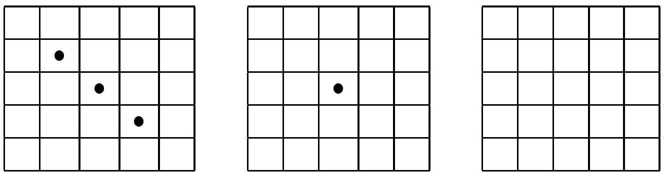
- **overlever** til neste generasjon hviss 2 eller 3 nabofelt har brikker
- **dør** hviss mer enn 3 eller mindre enn 2 nabofelt har brikker
- **fødes** i et tomt felt hviss nøyaktig 3 nabofelt har brikker

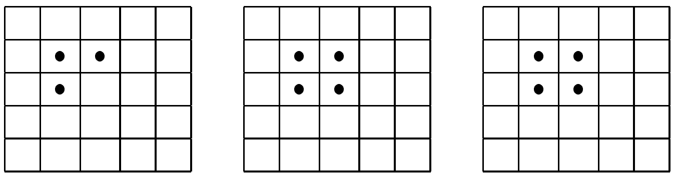
Nabofelt til et felt defineres her som de tilstøttende – vertikalt, horisontalt eller diagonalt – felt, dvs. feltet $x : y$ i midten har følgende nabofelt (ingen felt er sin egen nabo)

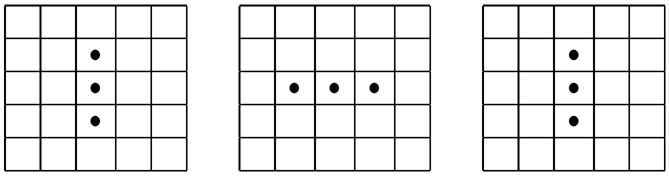
$-1 : -1$	$\_ : -1$	$+1 : -1$
$-1 : \_$	$x : y$	$+1 : \_$
$-1 : +1$	$\_ : +1$	$+1 : +1$

Hvis noen av disse feltene havner utenfor brettet, regnes de ikke med bland naboer.

Her er tre eksempler på spillforløpet fra venstre til høgre – i det første dør alle etter andre generasjon; i det andre, dannes det en stabil formasjone; i det tredje oppnåes en udødelig vekselvirkning mellom en horisontal og en vertikal 3-brikker formasjon:

1) 

2) 

3) 

Et *Step* i denne algoritme skal, naturlig nok, beregne og vise neste generasjon.

5 Løsning

5.1 Moduler og grensesnitt

I første omgang ignorer de konkrete algoritmene som skal implementeres og lag en struktur for hele programmet. Du skal spesifisere passende grensesnitt for de forskjellige moduler du skal bruke. De kan defineres som `interface` eller, om det er hensiktsmessig, som `abstract class`. Dokumentasjon skal også inneholde forklaring på hvordan modulene er tenkt å samarbeide.

Denne delen skal også løse alt som har med bruker-grensesnitt å gjøre.

5.2 Algoritmer

For hver konkret algoritme som skal implementeres:

- si kort hvordan den er implementert,
- om du bruker noen tilleggsdatastruktur
- hva er tidskompleksiteten av selve kjernelagoritme, samt hele implementasjonen (som for en enkel algoritme kan kreve flere hjelpeoperasjoner); for de nye algoritmene begrunn analyse av tidskompleksitet.

Implementer alle algoritmene.

6 Gitte moduler

Du har gitt to moduler: `gridFrame` og `ioFrame`. Den siste kan du finne nyttig i design og implementasjon av ditt eget grafisk brukergrensesnitt for innlesing av enkle parametere, utskrift til bruker osv. Ditt hovedprogram vil i så fall utvide og spesialisere denne klassen. Alternativt kan du lage hele brukergrensesnitt selv. Dens spesifikasjon følger i 6.2.

Hovedmodulen – som du skal bruke for visulaisering – er `gridFrame` og vi beskriver den nå.

6.1 Klassen `gridFrame`

Vi beskriver først i 6.1.1 hvordan et objekt av denne klassen opptrer for sluttbruker. Deretter, 6.1.2, beskriver vi grensesnitt som kan benyttes i programmering.

6.1.1 Bruker grensesnitt

Et `gridFrame` objekt viser for sluttbruker et vindu med en matrise samt tre menyer. Menyene brukes av sluttbruker for å kontrollere display. Man kan velge

1. fra meny 'Grid':
   - størrelse på enkle ruter i matrisen
   - 'On' eller 'Off' – man kan vise/skjule linjene som skiller rader og kolonner
2. fra meny 'Font':
   - 'color' eller 'numeric' – brikker vises enten som fargete sirkler eller som tall
   - skriftstørrelse (relevant kun for 'numeric')
3. fra meny 'Color' (relevant kun for 'color'):
   - 'Show' viser korrespondanse mellom tallene 1–9 og farger
   - denne kan forandres:ved å klikke på en farge, merkes den med $*$; ved å klikke på en annen farge (med CTRL-tasten nede) vil den andre fargen og den merket med $*$ bytte plass
   - 'Default' – bringer tilbake opprinnelige korrespondansen

Man kan bruke opp til 9 farger og 'default' korrespondanse er gitt ved:

tallet	–	farge
0	–	ingen (samme som bakgrunn)
1	–	white
2	–	black
3	–	blue
4	–	green
5	–	red
6	–	yellow
7	–	orange
8	–	pink
9	–	cyan

Dvs. er **`gf`** i 'color' modus, vil et kall til **`gf.showBead(1,3,5)`** resultere i at en rød sirkel plasseres i rad 1 kolonne 3. (Er siste argumentet støre enn 9, f.eks. **`gf.showBead(1,3,21)`**, vil man få en 'cyan' sirkel.) Er **`gf`** i 'numeric' modus, vil tallet 5 (21) bli plassert der.

Man veksler manuelt mellom 'color'/'numeric' modus fra en meny. Legg merke at, i oppgave 2.2, skal man kunne veksle mellom forskjellige algoritmer som kan innebære at man vil veksle display. Dette må brukeren gjøre selv fra meny.

6.1.2 Programmeringsgrensesnitt

En hovedobservasjon ang. `gridFrame` er at den ikke har noen intern datastruktur for å holde rede på den matrisen den viser på skjermen. (Følgelig, har den ingen 're-fresh' funksjon denne må programmeres fra et objekt som har en slik matrise.) Grensesnitt-metodene tillater programmereren å plassere/fjerne brikker av en ønsket verdi (numerisk verdi – om disse vises som tall eller som farger bestemmes av sluttbrukeren fra meny). Dermed, en modul som holder en matrise og kjører en algoritme på denne, vil måtte kalle slike oppdateringsmetoder fra `gridFrame` hver gang tilstanden i matrisen forandres (og skal vises for sluttbrukeren).

På den andre siden, implementerer `gridFrame` `MouseListener`, spesielt metoder `mouseClicked, mousePressed, mouseReleased`. Men det eneste denne implementasjonen gjør er å sjekke om hendelsene fant sted innenfor matrisen på skjermen og, i så fall, sende de til tilsvarende metoder i det kallende objektet som selv må være en implementasjon av `MouseListener`.

Dokumentasjon av programmeringsgrensesnittet til `gridGrame` følger med oppgaven.

6.2 Klassen `ioFrame`

Klassen gir bare muligheter for meget forenklet oppsett av et vindu med knapper og felter. (Det skal være tilstrekkelig for oss, så du kan bruke denne klassen for å forenkle arbeidet ditt. Men du står fritt til å lage ditt egen brukergrensesnitt dersom du ønsker det.) Den setter opp et vindu med en knapp 'Quit' – med forventet virkning. Den `implements ActionListener` men bare i den grad at den avslutter kjøring når brukeren trykker på 'Quit'-knappen. Alle andre ønskelige aksjoner må implementeres i utvidelser av klassen.

Dokumentasjon av klassen `ioFram` følger med oppgaven.

7 Har du lyst, så har du lov...

å inkludere flere algoritmer.

7.1 Ulam's 'Expand'

Dette spillet ligner litt på Conway's 'Life' men har andre regler som styrer utvikling fra generasjon til generasjon. Nabofelt her er bare de vertikale og horisontale naboer, slik de var definert i 4.1.

Spillet (av polsk matematiker Stanisław Ulam) går som følger: i generasjon n fødes det en ny brikke i et tomt felt hviss nøyaktig ett nabofelt hadde en brikke i generasjon $n-1$; også alle brikker fra generasjon $n-2$ dør i generasjon n. Mao., generasjon

n: et eller annet

n+1: det fødes en ny brikke i et tomt felt hvis nøyaktig ett nabofelt hade en brikke i generasjon n; alle brikker fra generasjon $n-1$ dør

n+2: tilsv.: en brikke fødes hvis ett nabofelt hadde en brikke i generasjon $n+1$; alle brikker fra generasjon n dør

osv.

Her følger 4 første steg fra en enkel brikke (vi tenker oss at brettet er større enn tegnet her; $\bullet$ viser nye brikker og $\circ$ brikker fra forrige generasjonen som skal dø i neste generasjon)