Inlämningsuppgift – Sudoku

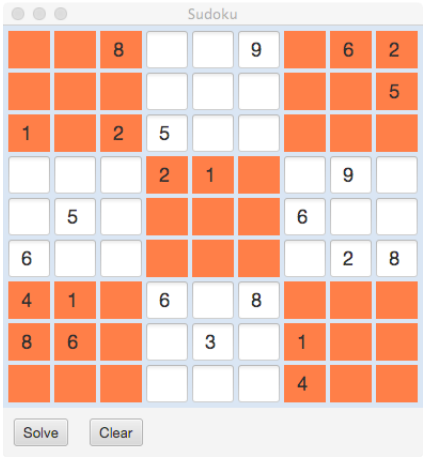
Uppgiften går ut på att skriva ett program som löser Sudoku. Användare skriver in siffror i några rutor och kan sedan begära att få en lösning. Lösningen ska beräknas med rekursiv teknik med s.k. backtracking. Programmet ska ha ett grafiskt användargränssnitt.

1 Problemet

Sudoku är ett ”pussel” som består av 9*9 rutor. Några rutor är från början ifyllda med siffror mellan 1 och 9. Uppgiften går ut på att hitta en lösning som uppfyller följande krav:

- Varje ruta är fylld med en siffra mellan 1 och 9.
- Varje rad, vågrät och lodrät, innehåller siffrorna 1–9.
- Varje ”region” innehåller också siffrorna 1–9. Med region avses här en grupp av nio rutor motsvarande de grupper som färgats ljusa repektive mörka i figur 1.
- Ingen av siffrorna förekommer mer än en gång per rad, kolumn eller region.

Exempel finns i figur 1 och 2. Figur 1 visar vilka rutor som från början var ifyllda och figur 2 visar en lösning.



Figur 1: Sudoku med ett antal från början ifyllda rutor.

Figur 2: Lösning till Sudokun i figur 1.

2 Programskrivning

Programmet består av två delar. En del som utgör en modell av sudokun och en del som utgör ett grafiskt användargränssnitt.

Det viktigaste är modellen. Denna del bör lösas och testas först. Därefter kan man implementera det grafiska användargränssnittet. Innan man har ett sådant kan man testa modellen, t.ex med JUnit genom att skapa sudokurutnät med vissa rutor givna och sedan anropa den metod i modellen som löser problemet.

2.1 Modellen för Sudoku

Ett Sudoku representeras lämpligen av en klass som har en heltalsmatris motsvarande sudokuns rutnät som attribut.

I klassen ska det bl.a. finnas en konstruktor och metoder för att sätta repektive hämta värdet för en ruta.

Den viktigaste metoden i klassen är den som löser problemet. Utgående från ett delvis fyllt rutnät ska denna metod hitta en lösning eller meddela att ingen lösning finns. Problemet ska lösas rekursivt med s.k. backtrackingteknik.

Backtracking kan användas för problem där man genom att systematiskt prova ett begränsat antal möjligheter kan konstruera en lösning. Ett exempel finns i boken, avsnitt 5.6. Här gäller det att finna en väg genom en labyrint. En rekursiv lösning bygger på följande resonemang: när man befinner sig i en viss position (ruta) i labyrinten så kan man hitta en utväg ur labyrinten om man från någon av de intilliggande rutorna kan hitta en utväg. Man har fyra möjligheter att gå vidare från en ruta: att gå till närmaste ruta högerut, vänsterut, uppåt eller nedåt. Dessa möjligheter undersöks i tur och ordning och man gör rekursiva anrop som undersöker om det finns en utväg från någon av dessa grannar. Om något av dessa anrop hittar en utväg är vi klara. Annars har vi ingen utväg (lösning) från den ruta vi befann oss i. Då backar vi tillbaks till den ruta vi närmast kom ifrån. Om det finns fler outforskade alternativ från denna väljer vi ett av dessa. Annars backar man en ruta till etc. Denna backning (backtracking) sköts automatiskt av rekursionen genom att man återvänder till anropande upplaga av den rekursiva metoden. På detta sätt kommer man systematiskt att utforska alla alternativ tills man hittar ett som lyckas eller kan konstatera att alla misslyckas.

Läs exemplet i boken. Observera dock att detta är mer komplicerat än vårt sudokuproblem. I en labyrint kan man hamna i en cykel genom att man på olika vägar kommer tillbaks till samma ruta. Detta hindras genom att man i lösningen markerar rutor som besökta. I sudokun kommer detta inte att behövas. Här kommer vi alltid att gå framåt element efter element rad efter rad i vårt sökande efter lösning.

2.2 Rekursiv lösning för Sudoku

När användare matat in siffror i rutor och begärt att få se en lösning kan olika fall inträffa:

- Det saknas lösning. Då ska programmet upptäcka och meddela detta.
- Det finns flera lösningar. Då räcker det att programmet finner en lösning och presenterar denna.
- Det finns en enda lösning. Då ska denna hittas och presenteras.

Alla dessa fall kan hanteras av en rekursiv metod som använder backtracking. Metoden kan ha följande signatur:

```
private boolean solve(int i, int j);
```

För att lösa problemet startar man på rutan längst uppe till vänster, (0,0), vilket motsvarar ett anrop `solve(0,0)`. För en enskild ruta (nedan kallad aktuell ruta) finns det två fall:

1. Aktuell ruta är inte från början fylld (av användaren). Då provar man i tur och ordning att fylla den med något av talen 1..9. För varje sådant val kontrollerar man först att det är möjligt med hänsyn till reglerna för Sudoku. Om det är möjligt fyller man i rutan och gör ett rekursivt anrop för nästa ruta. Med nästa ruta avses här rutan till höger om aktuell ruta eller (om aktuell ruta är den sista på en rad) första rutan på nästa rad. Om det inte går att fylla aktuell ruta med något av alternativen eller om de rekursiva anropen returnerar `false` för alla de alternativ man provar, så markeras aktuell ruta som som ej ifylld (backtracking) och man returnerar `false`. Om däremot något av de rekursiva anropen returnerar `true` så har man hittat en lösning och kan returnera `true`.

2. Aktuell ruta är från början fylld (av användaren). Då ska vi inte prova några alternativ utan bara kontrollera att det som är ifyllt är ok enligt reglerna. Om så är fallet görs ett rekursivt anrop för nästa ruta och resultatet av detta returneras. Om den ifyllda rutan däremot inte uppfyller villkoren har man misslyckats och returnerar `false`.

För båda alternativen ovan gäller att om ”nästa ruta” inte finns (d.v.s. vi har gått igenom hela rutnätet) så har vi lyckats fylla i alla rutor och kan returnera `true`.

2.3 Grafiskt användargränssnitt

Programmet ska ha ett grafiskt användargränssnitt. Se figurerna 1 och 2 som exempel på hur det kan se ut. När programmet startar ska ett tomt rutnät med 9*9 rutor samt två knappar visas.

Rutorna kan var av typen `TextField` med plats för ett tecken vardera. Dessa textfält placeras på en panel av typen `TilePane` (som i sin tur placeras i en annan panel). För att tydligt markera olika regioner kan olika bakgrundsfärg användas för textfälten.

Användare skriver in siffror i ett antal rutor. Därefter kan man be att få se en lösning genom att trycka på knappen ”Solve”. Knappen ”Clear” ska ta bort alla siffror från alla rutor.

I den lyssnaren som kopplas till knappen ”Solve” ser man till att läsa av alla textfälten och föra över motsvarande värden till modellen. Därefter anropas modellens `solve`-metod. Om denna returnerar `true` hämtar man alla rutornas värden från modellen och visar dessa i motsvarande textfält. Annars visas ett dialogfönster där det anges att ingen lösning finns. Lyssnaren kopplad till knappen ”Clear” tömmer textfälten i vyn.

Programmet ska vara bekvämt att använda. Det ska inte krascha om användaren skriver in konstiga värden i rutorna.