En variant av inlämningsuppgiften Sudoku, avsedd för grupper om tre

Beskrivning

Denna uppgift är en variant av inlämningsuppgiften Sudoku, anpassad för grupper som vill jobba tre och tre. Kärnan i uppgiften, att interaktivt kunna lösa sudokuproblem, finns kvar. Anvisningarna för den ordinarie uppgiften vad gäller denna del gäller alltså även för den utökade varianten. Det som ändrats/tillkommit beskrivs nedan.

1 Ändringar/tillägg till ordinarie uppgift om Sudoku

En funktion skall skrivas som löser ett sudokuproblem. Utgående från ett delvis fyllt rutnät ska denna funktion 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 ”Koffman E.B., Wolfgang P. Data Structures, Abstractions and Design Using Java. 2 Ed.”, avsnitt 7.6. 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.

1.1 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 funktion som använder backtracking. Metoden kan ha följande signatur:

```
bool solve(int i = 0, int j = 0);
```

För att lösa problemet startar man på rutan längst uppe till vänster, (0,0), vilket motsvarar ett anrop `solve()`. 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`.

1.2 Ändringar i det grafiska användargränssnittet

Ett tryck på knappen ”Check” skall även göra ett anrop till `solve()` för att avgöra om någon lösning finns till den inmatade ställningen.
Funktionen `solve()` skall också användas för att implementera två nya knappkommandon:

1. ”Solve” betyder att avvändaren ger upp och vill ha lösningen visad. Funktionen `solve()` appliceras på den inmatade ställningen. Om en lösning finns så fylls de tomma rutorna i med motsvarande siffror från lösningen. Om lösning saknas så skall detta meddelas till användaren.

2. ”Hint” betyder att användaren fastnat i lösandet och vill ha hjälp att komma vidare. Applicera `solve()` på den inmatade ställningen för att hitta en lösning. Om en lösning finns så fylls en av de tomma rutorna i med motsvarande siffra från lösningen. Om lösning saknas så skall detta meddelas till användaren.