Sudoku Solver

Mattias Cederlund

29-11-2012

1 Uppgiften

Uppgiften är att skapa ett program som löser sudoku-pussel genom propagering och sökning. Sökaren ska utgå från en gissning och propagera vidare till de andra fälten vilka val som gjorts icke möjliga. Med hjälp av recursive backtracking kommer programmet att revidera sin gissning ifall att gissningen inte ledde till en lösning. Ett sudoku-pussel är löst då alla fält har en siffra, samt att det endst finns en siffra av varje sort 1-9 i varje rad, kolumn och 3x3-ruta.

2 Ansatts

2.1 Implementera fält

Den första uppgiften bestod i att implementera en klass Field som håller ett bitset för att hålla koll på vilka rutor i sudokut som är satta till en viss siffra och vilka siffror som är möjliga val för given ruta. Klassen innehåller en int _digits som har bitar som representerar vilka siffror som är möjliga att placera i en ruta. Har _digits endast en bit som är en etta är den rutan satt till ettans bitposition. Klassen innehåller dessutom metoder för att hantera detta.

En metod assigned() returnerar true om det endast finns en etta i _digits. Det kollar den genom att loopa igenom de 9 första positionerna och kolla om den hittar en första etta. Hittas denna samt en till returneras false, annars true. Metoden value() kollar vidare ifall ett fält är assignat vilken siffra som står i fältet genom att loopa på samma sätt genom de första 9 positionerna och kollar om det finns en etta där. Är så fallet returneras det index ettan hittats på, alltså vilken siffra som ska stå i fältet.

Metoden in(digit d) undersöker om en viss siffra är inkluderad i setet, alltså om den är ett möjligt val för rutan. Detta kollas genom att man ANDar _digits med en etta shiftad till given position av d och kollar om detta är nollskilt. Är så fallet är d en möjlig siffra för fältet. Vidare finns metoden prune(digit d) som tar bort ettan på position d genom att ANDa

_digits med inversen av en etta skiftat till positionen d, och därmed sätter att den inte är ett möjligt val för fältet. Metoden assign(d) sätter en etta på given position d och bestämmer därmed att det står en siffra d i fältet. Både prune och assign kan endast köras givet att d är ett giltigt val, kontrollerat av in-metoden.

2.2 Implementera propagering

2.2.1 Konstruktorer, assignment operator, printing

I klassen Board som symboliserar ett sudokubräde med sina 9x9 rutor skapar vi en konstruktor där vi skapar Fields för alla 9x9. Rutornas pekare lagras i en primitiv tvådimensionell array fs, där vi loopar igenom indexen och kallar på Fields konstruktor.

Copy-konstruktorn loopar på samma sätt igenom alla 9x9 fält i fs, och skapar nya fält med inputfältet som argument. Kompilatorn kommer då att lägga upp en default-konstruktor som skapar ett fält med samma värde som fältet givet som argument.

Assignment-operatorn undersöker först om båda sidor är lika, isåfall returneras en pekare till this. Annars används delete på fs och sedan funktionen std::copy som kopierar argumentets fs till sin nya plats.

2.2.2 Assign

Metoden assign(int fx, int fy, digit d) ska om det är möjligt assigna d till det fält fs[fx][fy] pekar på. Vi kollar först om d är ett möjligt värde för fältet med hjälp av in(d), om inte returneras false. Om det är möjligt kollar vi om fältet redan är assignat till d. Är så fallet returneras true direkt. Annars assignar vi fältet till d och anropar propagate(fx, fy) och returnerar sedan true.

2.2.3 Propagate

Metoden propagate(int fx, int fy) ska när vi assignat en ruta se till att samma siffra inte ska vara ett möjligt val på de rutor på samma rad, kolumn och 3x3-fält. Givet positionen av utgångsrutan kollar vi vilket värde den blivit assignad till med value(). Sedan loopar vi från 1 till 9 i först kolumn och anropar prune på de olika rutorna, bortsett frå utgångsrutan. Sedan gör vi likadant för rader. På så sätt har vi gått igenom alla fält både horisontellt och vertikalt från utgångsrutan och sett till att siffran inte kan assignas där.

För att göra detsamma för alla fält i 3x3-rutan så beräknar vi vilken ruta vi är i genom att spara ner (fx%3) och detsamma för fy. När vi nu loopar från 0 till 3 för både rad och kolumn anropar vi prune på ruta fx-(fx%3)+i, fy-(fy%3)+k där i och k är index. På så sätt startar vi på rutan längst

upp till vänster i respektiva 3x3-ruta. T.ex. ruta [5][6] tillhör 3x3-rutan med översta vänstra hörn i [5-2][6-0], alltså [3][6], dvs rutan längst ner i mitten.

2.2.4 Prune

Metoden prune(int fx, int fy, digit d) ska ta bort möjligheten för en viss siffra att stå i givet fält. Först kontrolleras om d är ett möjligt val för fältet genom metoden in(d). Är så fallet anropar vi prune(d) på givet fält. Sedan kollar vi om fältet nu efter att vi tagit bort en möjlig siffra endast har en siffra kvar och därmed är assignad. Är så fallet anropar vi propagate(fx, fy) och returnerar true. Om d inte var ett möjligt val för fältet returneras false.

2.3 Search: Copy

Funktionen som letar efter en lösning heter search() och returnerar en pekare till ett Board. Antingen till en lösning eller till NULL om ingen lösning hittats. Funktionen letar efter en lösning till sudokut med hjälp av recursive backtracking. Alltså görs en gissning och om den inte slutar i en lösning backar algoritmen och gör en annorlunda gissning.

Vi loopar igenom fs-matrisen och kollar om ett field inte är assignat. Isåfall loopar vi från 1 till 9 där vi först skapar en kopia av brädet. På denna kopia kollar vi om en gissning med siffran given av den inre loopen är möjlig med in(d). Isåfall assignar vi fältet och sätter en pekare till ett board vara search() på kopian av bordet; vi gör ett rekursivt anrop på det nya bordet för att göra en andra gissning. När search() returnerar kollar vi ifall om den inte är null. Är så fallet returnerar vi solution. Är det däremot null kommer kopian av brädet att deletas och en ny skapas med en gissning på ett annat nummer på fältet.

search() returnerar en Board-pekare till this när den gått igenom alla fields och dessa är assignade. Resultatet blir att alla rekursiva recalls returnerar och ursprungsanropet returneras med en lösning till sudokut.

2.4 Search: Trailing

2.4.1 Klassen Trail

En trail är i princip en länkad lista som innehåller trails för att spara ner assigns och prunes; när informationen om ett fält förändras. Den innehåller även marks, som markerar när sista gången sök-funktionen gjorde sin senaste assign. Som noder används TrailEntrys som innehåller en pekare till föregående entry. Trail har en pekare till den senaste noden och vida den kan noderna itereras över.

Metoden undo() i Trail ska återställa värden sparade i trailen fram tills nästa mark. Ett mark är en entry utan pekare till ett fält. undo() loopar tills NULL påträffas, alltså tills det inte finns några entrys. Först undersöker man

om toppen-trailen är en mark, isåfall ska toppen sättas till markens next-pekare och sedan breakas loopen. Om entryn inte är en mark ska man kalla på entryns undo()-metod, som återställer dess fälts värde. Sedan sätter man pekaren till toppen-trailen att peka på nuvarande toppen-trails next osv.

2.4.2 Sökning med trail

Sökningen med en trail går till i princip likadant som sökningen med copy. Board-metoderna assign och prune ska innan man anropar fältets assign- eller prune-metoder skapa en trail innehållande det nuvarande värdet. Sökningen är annars snarlik copy. Skillnaden är att innan sökmetoden gör en gissning och assignar ett fält ska den lägga en mark till trailen. Sedan anropas search(t) igen rekursivt och sparas ner till en boolean. När searchmetoden returnerar antingen false när den inte lyckats assigna något värde till fältet, eller true när alla fält redan är assignade kommer detta kollas. Ifall det är true så kommer hela rekursionskedjan att returnera true och vi har en lösning. Vid en false-retur anropas trailens undo-metod som kommer återställa sudoku-brädet till hur det såg ut innan gissningen, och därmed göra en ny gissning möjlig.

3 Utvärdering

Både sökning copy och trail returnerar giltiga lösningar på de pussel jag testat. Trail-metoden verkar dock ha minnesläckor, medan copy inte har det. Vid repeterade försök lyckas trail lösa 1000 utan att gå över 3.5Mb i minnesanvändning, men hela tiden ökande. Trail är dock mer än dubbelt så snabb på att hitta en lösning som copy är.

4 Sammanfattning

Bortsett från småfel så har labben inte ställt till med så mycket problem. Ett återkommande problem var dock asserts. För att slippa dessa behöver vi hela tiden kolla om ett fält är assignat innan vi anropar value(), vilket kunde konstruerats annorlunda. Man kunde returnerat t.ex. -1 om fältet man ville ha value på inte var assignat och sedan utveckla resten utifrån det. Då hade assigned()-metoden varit överflödig.

Copy-konstruktorn skapade lite problem då jag från början endast gjorde en shallow-copy med hjälp av std::copy. Lösningen loopar nu istället genom arrayen som lagrar pekare till fälten och skapar nya fält med kompilatorns automatiskt genererade copy-konstruktor för Fields som tar ett Field som argument.

För att inte copy skulle få några minnesläckor la jag till en destruktor för board som avallokerar dess fält.