Att lösa Sudoku

Problem av matematisk natur kan ofta snabbt lösas med dator och lite programmering. Därför har sällan matematiska pussel som Sudoku lockat att lösa ”för hand”, men jag har länge tänkt att det vore väl enkelt att lösa med några rader kod som tar fram lösningen.

Sagt och gjort, men matematiskt är det också intressant när ”brute force”-algoritmer där helt enkelt alla möjligheter provas ger så många kombinationer att det även för en snabb dator med sina miljarder klockcykler per sekund tar oerhört lång tid att prova sig fram – grunden för säker kryptering är just att det finns så oerhört många kombinationer att prova att inte ens en dator inom rimlig tid kan knäcka koden.

För Sudoku handlar det i standardfallet om 9 x 9 rutor med 9 möjliga värden, vilket kan uppfattas som nära ett tal med 81 siffror (förutom att siffran 0 inte får användas). Rent matematiskt motsvarar det 9^81 = 1,97 x 10^77, alltså ett tal med 78 siffror. Med tanke på att en miljard skrivs 1 x 10^9, en miljard miljarder 1 x 10^18 så är det oerhört många kombinationer att prova, även för en dator.

Dock finns många enkla regler att följa i Sudoku som minskar ner mängden alternativ att ens överväga. Att skriva 1:or på alla lediga platser är onödigt att ens överväga – att ens skriva två 1:or på samma rad ger ingen lösning. Så även ”brute force”-algoritmen kan kapa bort många kombinationer tidigt genom att utvärdera om ett alternativ kan leda till en lösning, eller ej.

Ett enkelt försök att lösa Sudoku utformas med:

+ en matris med 9 x 9 värden, 0 för ledigt och 1-9 för valda värden att testa
+ en funktion som börjar testa på första rutan med en 1:a och verifiera att detta är ett giltigt värde
+ är det giltigt anropar funktionen sig själv (rekursion) för att testa nästa ruta
+ vid retur från det rekursiva anropet testas en 2:a och så vidare
+ till slut har man nått den sista rutan och bör ha en lösning

Från start har matrisen några värden ifyllda, enligt det givna Sudoku-problemets siffror – färre ifyllda på svårare Sudoku, fler på de lite enklare, men funktionen hanterar en redan ifylld ruta genom att inte prova 1 till 9 utan bara kolla att det fortfarande är en giltig lösning.

För att enkelt koda detta väljs script-språket php (finns förinstallerat för kommandorad på Mac, enkelt att installera i Windows och utgör grunden för mycket av webbens verktyg som WordPress, Joomla, Drupal med flera). Koden utformas som en klass (objektorienterad programmering) med data för själva Sudoku-matrisen samt metoder för att kolla giltighet för en ruta och den rekursiva metoden för att lösa ett steg i taget.

Index i matrisen startar på 0 så rad och kolumn-värden går från 0 till 8 (mot de för människor mer naturliga 1 till 9).

Metoden Solve (börjar nedtill på sid 3)

Koden inleder med att räkna fram nästa ruta genom att öka kolumnvärdet med 1 och går man över kanten nollställs kolumnen och raden ökas med ett – från ($r,$c) till ($rn,$cn).

Därefter kontrolleras om det redan finns ett värde på den givna platsen, isåfall kontrolleras giltigheten med metoden Verify och är det giltigt sker rekursivt anrop (metoden anropar sig själv) med argument för nästa ruta.

Finns inget värde (tom ruta är 0) så startas en loop för att prova varje möjlig siffra på den givna platsen och är siffran giltig enligt metoden Verify sker rekursivt anrop för nästa ruta.

Metoden Verify (mitt på sid 3)

Som argument tar funktionen rad och kolumn att kontrollera giltighet för siffra i matrisen. Enligt reglerna för Sudoku får inte samma siffra finnas på samma rad eller i samma kolumn, ej heller i den mindre 3x3-matris siffran finns inom.

Med en loop från 1 till 9 kontrolleras att inte samma siffra som på given plats finns på raden, därefter i kolumnen.

För den mindre 3x3-rutan beräknas först rutans startrad och startkolumn (0, 3 eller 6), därefter kontrolleras genom nästade loopar att inte samma siffra som på given plats finns på annan plats inom kvadraten.

Metoden returnerar falskt så fort samma siffra hittas och klaras alla tester returneras till slut sant.

Övrig kod i klassen

En konstruktor som initierar matrisen till 0-värden samt en metod Show som visar matrisen på ett enkelt sätt är också implementerade. En räknare count håller reda på hur många anrop som gjorts till metoden Verify.

Huvudprogrammet (nedre halvan av sid 4)

Här skapas ett objekt av klassen SudokuSolver, så sätts startvärden genom att tilldela 9 rader med 9 siffervärden (matas in från exempelvis tidning), så anropas metoden Solve för första rutan (0,0).

Utfall (utmatning upptill sid 5)

Hur fungerar koden att köra då? Jotack, över förväntan, det svåra problemet ur SvD 2016-03-19 där 26 siffror av 81 siffror givits tar ca 1 sekund att lösa och istället för de många miljarder alternativen så har metoden Verify kontrollerat 156 380 möjliga siffror och den enda giltiga lösningen hittades efter 116 070 kontroller. Den enklare rutan med 36 givna siffror löstes på blott 0,1 sekund.

Utökning

I standardform gäller storleken 9x9 med siffror 1-9 i varje ruta, men större Sudoku kan tänkas med i grunden samma regler. 16x16 och 25x25 exempelvis, varvid siffrorna 1-9 utökas till 1-16 resp. 1-25. För att slippa 2-ställiga tal kan bokstäver a-g resp. a-p användas vid presentation av matrisen.

Koden justeras så förekomsten av siffran 3 ersätts med SIZE och 9 med SQUARE – därefter kan SIZE definieras till 4 eller 5 så fungerar exakt samma kod även för större problem.

Hade koden varit långsam kunde man portera till ett kompilerande språk som C/C++ och använda flera processorkärnor genom att dela problemet i flera delar, men det behövdes nu inte i detta fall.

Koden

141 rader PHP-kod (varav 32 rader enbart kommentar och 18 tomrader så *under 100 rader kod*):

```php
<?php

// Definiera storlek på lilla rutan, 3, 4, 5 eller...
define('SIZE', 3);
define('SQUARE', SIZE*SIZE);

// Klass med data och metoder
class SudokuSolver {
  public $m=array();     // Matrisen med siffror, 2-dimensionell vektor
  public $count=0;       // Antal kontroller i Verify
```

```
// Konstruktor som initierar data - fyll matrisen med nollor
function __construct() {
  for($r=0; $r<SQUARE; $r++)
    for($c=0; $c<SQUARE; $c++)
      $this->m[$r][$c]=0;
}

// Visa matrisen
function Show() {
  // Symboler - siffror 0-9, sen bokstäver
  static $sym="0123456789abcdefghijklmnopqrstuvwxyz";
  //           0         1         2         3         4

  $set=0;    // Räkna antal satta värden
  for($r=0; $r<SQUARE; $r++) {
    for($c=0; $c<SQUARE; $c++) {
      echo ($c>0 && $c%SIZE==0 ? " " : "").$sym[$this->m[$r][$c]];
      if($this->m[$r][$c]>0) $set++;
    }
    echo "        ".(($r+1)%SIZE==0 ? "\n\n" : "\n");
  }
  // Ange andel löst om ej helt klar
  if($set<SQUARE*SQUARE)
    printf("Stat: %d (%1.1f%%) done\n", $set, (100.0*$set)/(SQUARE*SQUARE));
}

// Kolla giltighet för siffra i matrisen på rad $r, kolumn $c
function Verify($r, $c) {
  $this->count++;      // Räkna upp antal gjorda kontroller

  // Observera:
  // + vid kontroll av dublettvärden i rad, kolumn och lilla rutan
  //   jämförs med alla andra värden - utom på den egna platsen $r,$c
  // + värdet 0 för annan ledig ruta ger ingen dubblett då detta värde
  //   ej valts ännu, men kommer att väljas och kontrolleras då

  // Kolla efter dublett i samma rad
  for($i=0; $i<SQUARE; $i++)
    if($i!=$c && $this->m[$r][$i]==$this->m[$r][$c])
      return false;    // Ogiltig då samma siffra redan fanns på raden

  // Kolla efter dublett i samma kolumn
  for($i=0; $i<SQUARE; $i++)
    if($i!=$r && $this->m[$i][$c]==$this->m[$r][$c])
      return false;    // Ogiltig då samma siffra redan fanns i kolumnen

  // Kolla efter dublett i samma lilla ruta
  $rStart=SIZE*intval($r/SIZE);    // Rutans startrad
  $cStart=SIZE*intval($c/SIZE);    // Rutans startkolumn
  // Loopa över lilla rutans rader och kolumner
  for($rAdd=0; $rAdd<SIZE; $rAdd++)
    for($cAdd=0; $cAdd<SIZE; $cAdd++)
      if($rStart+$rAdd!=$r && $cStart+$cAdd!=$c &&
         $this->m[$rStart+$rAdd][$cStart+$cAdd]==$this->m[$r][$c])
        return false;  // Ogiltig då samma siffra redan fanns i lilla rutan

  // Returnera sant då ingen dublett fanns (ännu)
  return true;
}

// Lös Sudoku från rad $r, kolumn $c och framåt
function Solve($r, $c) {
  // Lite återkoppling på antal försök och var vi är
  if($this->count%10000==0)
```

```
      echo "\r".$this->count." at ($r,$c)";

    // Koll om färdig - då vi hamnat på raden under matrisen
    if($r>=SQUARE) {
      // Visa lösning
      echo "\rSolved after ".$this->count." checks:\n";
      $this->Show();
    } else {
      // Beräkna nästa ruta
      $rn=$r;
      $cn=$c+1;
      if($cn>=SQUARE) {
        $cn=0;
        $rn++;
      }
      // Kolla vad vi har på rad $r, kolumn $c
      if($this->m[$r][$c]>0) {
        // En redan given siffra, kolla om giltig och fortsätt isåfall
        if($this->Verify($r, $c))
          $this->Solve($rn, $cn);     // Fortsätt (rekursion)
      } else {
        // En ledig ruta, prova varje möjlig siffra och fortsätt om giltig
        for($digit=1; $digit<=SQUARE; $digit++) {
          $this->m[$r][$c]=$digit;    // Sätt siffra
          if($this->Verify($r, $c))
            $this->Solve($rn, $cn);   // Fortsätt (rekursion)
        }
        $this->m[$r][$c]=0;           // Återställ till ledig ruta
      }
    }
  }
}

//--- Huvudprogram ----------------------------------------

// Skapa instans av klassen
$sudo=new SudokuSolver();

// SvD 2016-03-19, svår
$sudo->m[0]=array(0, 0, 0, 0, 0, 0, 0, 0, 0);
$sudo->m[1]=array(6, 4, 0, 0, 1, 7, 0, 0, 0);
$sudo->m[1]=array(6, 4, 0, 0, 0, 0, 0, 0, 0);
$sudo->m[2]=array(0, 0, 0, 0, 2, 0, 3, 6, 1);
$sudo->m[3]=array(5, 0, 6, 0, 4, 0, 0, 0, 0);
$sudo->m[4]=array(0, 0, 0, 0, 0, 6, 4, 2, 0);
$sudo->m[5]=array(0, 0, 0, 2, 8, 0, 0, 0, 9);
$sudo->m[6]=array(0, 0, 0, 5, 0, 0, 7, 4, 0);
$sudo->m[7]=array(4, 0, 5, 0, 6, 9, 2, 0, 0);
$sudo->m[8]=array(8, 0, 0, 0, 0, 0, 0, 0, 0);

// Visa startläge
echo "Starting with:\n";
$sudo->Show();

// Lös från rad 0, kolumn 0
echo "Solving...\n";
$start=microtime(true);
$sudo->Solve(0, 0);
$stop=microtime(true);

// Rapportera körtid och totalt antal kontroller
printf("\rDone after ".$sudo->count." checks in %0.3fs.\n", $stop-$start);
?>
```

Utmatning

Vid körning visas:

```
Starting with:
000 000 000
640 017 000
000 020 361

506 040 000
000 006 420
000 280 009

000 500 740
405 069 200
800 000 000

Stat: 26 (32.1%) done
Solving...
Solved after 116070 checks:
312 695 874
648 317 592
957 824 361

526 943 187
189 756 423
734 281 659

291 538 746
475 169 238
863 472 915

Done after 156380 checks in 0.985s.
```

Värt att notera är att nya PHP version 7 är ca 2.25 ggr snabbare än PHP version 5 för denna kod.

Slutord

Med ovanstående exempel vill vi visa på möjligheten att lösa beräkningsproblem med en systematisk metod. ”Brute force” där alla möjliga kombinationer provas leder ofta till en kombinatorisk explosion med så många alternativ att inte ens en dator når ett resultat inom rimlig tid, men går det att tidigt eliminera alternativ så kan lösningen hittas inom rimlig tid - och i fallet Sudoku hittas lösningen på det svåra problemet på under sekunden!

Hör gärna av Dig om Du behöver hjälp att lösa något beräkningsproblem i Din verksamhet.

Fotnot: Koden togs fram på en Apple MacBook, en liten smidig men inte särskilt snabb dator, under lite väntetid på en flygplats.