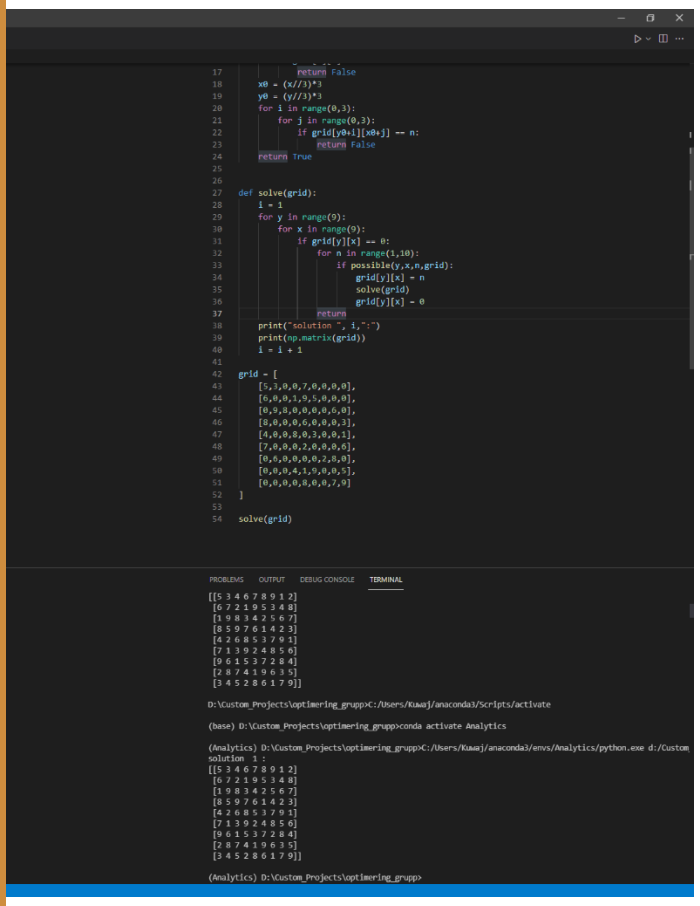
Grupparbete 2021 – Sudoku



December 10 2021

Arcada

Johan Penttinen, Fredrik Ståhl & Kristoffer Kuvaja Adolfsson

Introduktion

Uppgift

Inlämningsuppgiften går ut på att hitta ett verkligt problem som kan lösas med metoderna presenterad i kursen. Problemet kan utgå från en problemställning från en annan kurs eller ett problem som uppstår i vardagen/samhället. Systemet som studeranden bestämmer sig för kan vara ett ekonomiskt eller tekniskt system som skall optimeras. Vikt sätts vid att problemet är relevant och kan lösas med en känd metod. Data använt för de beräkningar som görs skall vara riktiga data eller ifall det är svårt att få tag i riktiga data så kan data estimeras, dock så att det är rimliga siffror. Man kan också göra ett teoretiskt arbete av typen ”ett script som tar ett godtyckligt travelling salesman problem och löser det med linjär programmering, med vilket man testar hur stora tsp problem som kan lösas med lp-solve”. Detta är bara ett exempel och man kan med fördel konsultera föreläsaren gällande ämne.

Inlämningsuppgiftens rapport skall vara minst 6 sidor lång. Den skall innehålla en del som beskriver problem-formuleringen (i text form samt någon typ av flow-chart) samt problemets relevans samt sammanhang. Det skall finnas en del som beskriver problemet matematiskt samt data för det, och en del som beskriver lösningsmetoden samt lösningen (även illustrerad grafiskt) för det data som använts.

Studeranden skall också skriva en sammanfattning om hur metoderna funkade och nyttan med dem i lösandet av det specifika problemet. Man får fritt använda illustration/grafer får att få fram viktiga poänger.

Arbetet presenterar under lektion den 10.12.2020. Håll en 10 minuters presentation (med hjälp av pptx eller dylika program) om ert problem och er lösning. OBS PRESENTATIONEN BEDÖMS OCH PÅVERKAR SLUTPOÄNGEN AV GW!!

Problemformulering

Vi har valt att söka sätt att lösa det logiska spelet sudoku med olika optimerings metoder. Vi vill skriva optimeringarna i Python och sedermera jämföra de olika metoderna i lösningsförmåga, hastighet och om möjligt komplexitet.

Vi väljer Python eftersom det är något vi arbetat med mycket tidigare och är bekanta med, samt att tiden är kort för projektet och alternativ som R därmed uteslöts eftersom det inte är bekant för oss.

Linjär och pmx är direkta metoder från kursen vi kommer nyttja men även en form av rekursiv (så kallad bakåtspårning/backtracking) samt verktyget PuLP i Python, vårt valda programmeringsspråk.

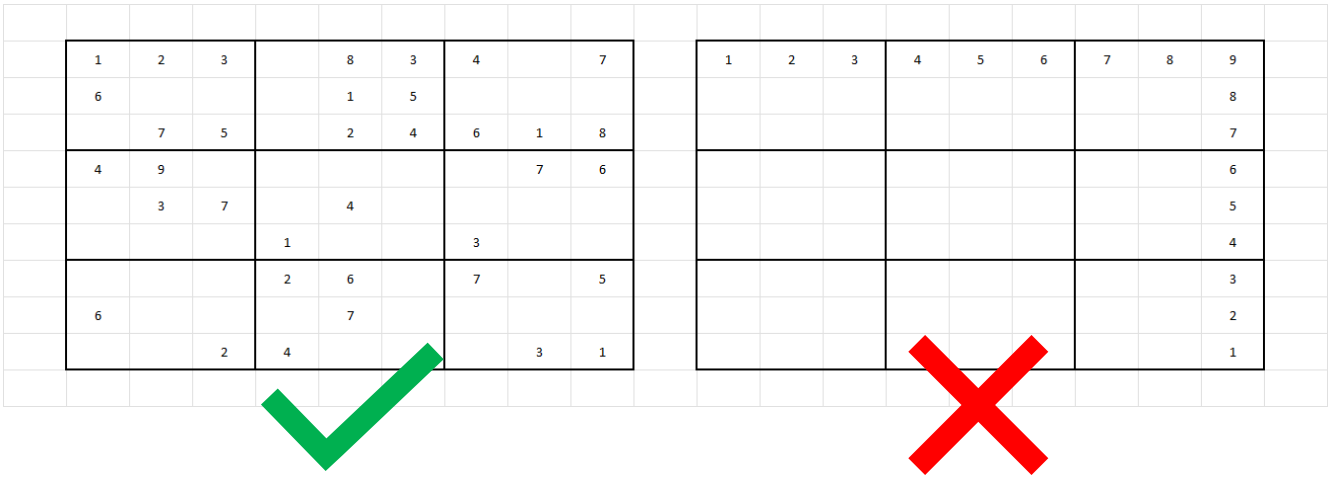
Sudoku definiering

Sudoku är ett numeriskt pussel som består av ett rutnät på 81 rutor som ska fyllas med siffrorna 1 till 9 utan att samma siffra får finnas i samma rad, kolumn eller låda, en låda består av de 9 3x3 rutnät som kan bildas inom sudokuns area, denna begränsning kallas även latinsk kvadrat.

Ett sudokuspel ska helst ha endast en lösning för att anses som bra eller äkta och fyllas med minst 17 siffror från början. Antalet förifyllda siffror avgör oftast svårighetsgraden för sudokupusslet. För närvarande kategoriseras oftast sudokun i fyra kategorier, lätta, mellansvåra, expert/svåra och djävulska (devilish).

Vi kommer välja sudokun i varierande svårighetsgrad med bias för första lämpliga alternativ som uppstår genom sökning på internet med Googles sökmotor, vilket leder oss till < https://sudoku.com/ > . Sudokuna ska

upplevas ha en sporadisk variation av slumpmässiga tal mellan 1 – 9 utplacerade på ett sådant sett att de inte anses manipulativa mot maskinkod, t.ex. ska de inte ha hela första raden eller kolumnen fyllda med resten av raderna tomma.



Vi kommer välja en handfull med lämpliga sudokun och spara dem i en Python modul som vi enkelt kan importera och nyttja i samtliga lösningar för projektet.

Lösningar

I följande kapitel kommer vi presentera våra olika metoder för att lösa sudoku.

Linjär programmering i lpsolve

Programmet för linjär programmering med blandade heltal (engelska MILP) kan lösa sudokun med rätt begränsningar och variabler. För att logiskt kunna utesluta valde vi att bygga upp våra variabler på följande sätt:

Sudokuna vi valt att arbeta med har 81 celler, dessa 81 celler har alla 9 möjligheter, en för varje siffra mellan 1 - 9. Därmed kommer vi definiera 729 (81 * 9) olika binära variabler. De binära variablerna beskrivs med x som generisk variabel term följt av numeriska värdet 1–9, därefter rad och kolumn även de med siffrorna 1–9, t.ex. en 3: a i första rutan, högst upp till vänster beskrivs med x311.

x	**3**	**1**	**1**
	Värde	Rad	Kolumn

Vi behöver sedan i huvudsak 4 generella begränsningar:

1. Varje cell får endast innehåll ett värde
2. Varje rad får endast innehåll ett av varje värde 1–9
3. Varje kolumn får endast innehåll ett av varje värde 1–9
4. Varje låda får endast innehåll ett av varje värde 1–9

För att begränsa varje cell till 1 värde adderar vi alla våra binänar värden för varje cell med begränsning att de tillsammans skall ha summan 1, t.ex. cellen i första raden, första kolumn:

x111 + x211 + x311 + x411 + x511 + x611 + x711 + x811 + x911 = 1

För att begränsa att en rad endast får ha ett av varje värde adderar vi samma värde för hela raden och begränsar det till 1, t.ex. endast ett värde 1 i den första raden:

x111 + x112 + x113 + x114 + x115 + x116 + x117 + x118 + x119 = 1

Samma logik gäller även för kolumner:

x111 + x121 + x131 + x141 + x151 + x161 + x171 + x181 + x191 = 1

Och för lådor:

x111 + x112 + x113 + x121 + x122 + x123 + x131 + x132 + x133 = 1

Varje huvudsaklig begränsning kommer därmed bestå av 81 begränsningar för totalt 324 begränsningar allt som allt.

Vi kan nu börja definiera ett sudoku, lyckligtvis är det relativt simpelt då ett sudoku kommer med färdiga celler ifyllda, vi behöver endast utesluta de cellerna från våra binära variabler och istället begränsa cellerna enligt sudokut, t.ex. är första radens första kolumn en 3: a sätter vi helt enkelt: x311 = 1.

Nu med all vår logik färdig gäller det bara att skriva in alla våra 1053 olika påståenden i lpsolve.

Vi valde att skriva ett Python program som skriver en lp-fil i stället, vi kombinerade det med våra färdiga sudokun.

Vi provade ett relativt enkelt sudoku, sudoku-easy 02:

	8	3		2	1			7
9	6		3		5	8	2	1
2	5	1			6		9	3
	4	8	1		2		7	
	2	9			4		3	
	3		7		8		4	
3	7				9		1	4
8	1		2	5	3		6	9
6	9		4	1	7		8	2

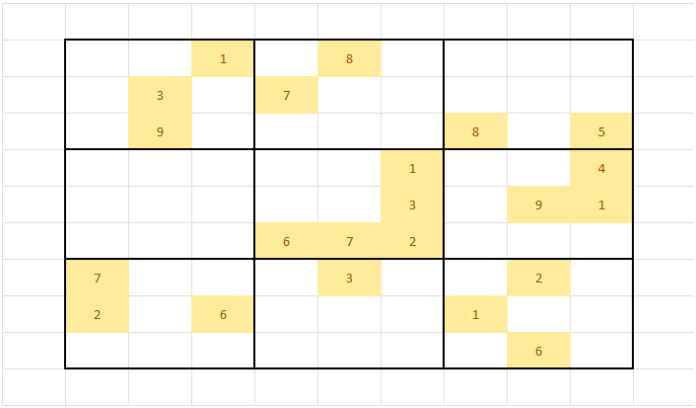
4	8	3	9	2	1	6	5	7
9	6	7	3	4	5	8	2	1
2	5	1	8	7	6	4	9	3
5	4	8	1	3	2	9	7	6
7	2	9	5	6	4	1	3	8
1	3	6	7	9	8	2	4	5
3	7	2	6	8	9	5	1	4
8	1	4	2	5	3	7	6	9
6	9	5	4	1	7	3	8	2

Resultat från solvern:

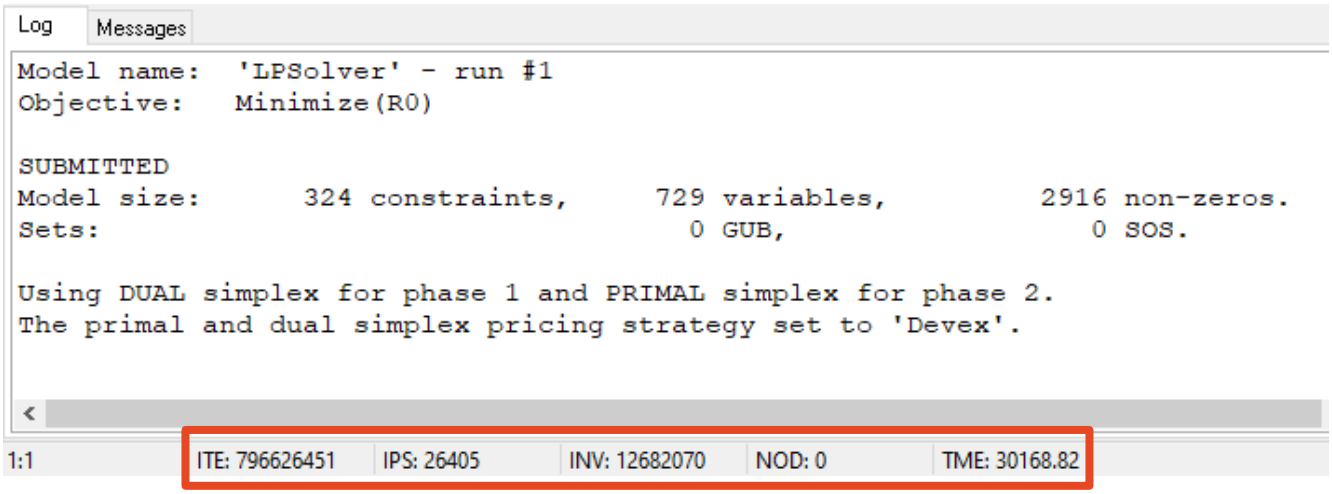
> In the total iteration count 342, 136 (39.8%) were bound flips.
> There were 3 refactorizations, 0 triggered by time and 3 by density.
> ... on average 68.7 major pivots per refactorization.
> The largest [LUSOL v2.2.1.0] fact(B) had 1075 NZ entries, 1.0x largest basis.
> The maximum B&B level was 1, 0.0x MIP order, 1 at the optimal solution.
> The constraint matrix inf-norm is 1, with a dynamic range of 1.
> Time to load data was 0.006 seconds, presolve used 0.005 seconds,
> ... 0.023 seconds in simplex solver, in total 0.034 seconds.

34ms för lpsolve för ett enkelt sudoku.

Vi provar ett svårare, sudoku-expert 04:



Dock efter över 8 timmar och nästan 800 miljoner iterationer stoppade vi solvern och gav upp på problemet.



Lpsolve har helt tydligt stött på en begränsning, vi antar att lpsolve försöker använda sin råa styrka (brute-force) för att linjärt helt enkelt prova alla kombinationer och även om problemet kanske är helt fullt ut möjligt att lösa finns det för många kombinationer i detta svårare sudoku för att lösa det inom en rimlig tid.

Första raden har 7 saknade celler med 7 olika siffror, det bör alltså finnas 181 440 permutationer (9*8*7*6*5*4*3). Multiplicera sedan detta med följande rad av samma kvantitet och vi kommer snabbt uppnå en storlek vi inte hinner lösa med lpsolve.

Linjär programmering i Python med PuLP

PuLP är ett program som kan läsa mps eller lp filer och kalla på olika linjära programmeringsmodeller för att lösa linjära problem. Programmet skiljer sig mestadels i syntaxen från lpsolve och inte i varken uppbyggnaden av problemet eller definieringen av begränsningarna. Det faktum att den har en annan logik i sitt val av lösare däremot påverkar resultaten, som vi förstår det har PuLP dessutom tillgång till ett breddare sortiment av modeller för lösningar.

Vi provar samma easy-sudoku 02 från tidigare och ser resultatet (resultatet är trunkerat samt rader innehållande hur många variabler som lästs och vilka filsystems vägar som använts är borttagna):

```
GLPSOL--GLPK LP/MIP Solver 5.0                    [[4 8 3 9 2 1 6 5 7]
Objective value =   0.000000000e+00               [9 6 7 3 4 5 8 2 1]
INTEGER OPTIMAL SOLUTION FOUND BY MIP PREPROCESSOR [2 5 1 8 7 6 4 9 3]
GLPSOL--GLPK LP/MIP Solver 5.0                    [5 4 8 1 3 2 9 7 6]
GLPK Integer Optimizer 5.0                        [7 2 9 5 6 4 1 3 8]
PROBLEM HAS NO PRIMAL FEASIBLE SOLUTION           [1 3 6 7 9 8 2 4 5]
Time:  93 ms                                      [3 7 2 6 8 9 5 1 4]
                                                  [8 1 4 2 5 3 7 6 9]
                                                  [6 9 5 4 1 7 3 8 2]]
```

Lösning ser ut att vara den samma som lpsolve kommit fram till och om vi sedan går vidare och provar det svårare sudokut kommer vi få resultatet:

```
GLPSOL--GLPK LP/MIP Solver 5.0
GLPK Integer Optimizer 5.0                        [[5 2 1 3 8 9 7 4 6]
 A: min|aij| =  1.000e+00  max|aij| =  1.000e+00  ratio =  1.000e+00  [4 3 8 7 6 5 9 1 2]
Problem data seem to be well scaled               [6 9 7 2 1 4 8 3 5]
Constructing initial basis...                     [3 6 5 8 9 1 2 7 4]
Size of triangular part is 136                    [8 7 2 4 5 3 6 9 1]
Solving LP relaxation...                          [1 4 9 6 7 2 3 5 8]
GLPK Simplex Optimizer 5.0                        [7 8 4 1 3 6 5 2 9]
208 rows, 192 columns, 768 non-zeros              [2 5 6 9 4 7 1 8 3]
      0: obj =   0.000000000e+00 inf =   3.000e+01 (27)  [9 1 3 5 2 8 4 6 7]]
     40: obj =   0.000000000e+00 inf =   0.000e+00 (0)
OPTIMAL LP SOLUTION FOUND
Integer optimization begins...
Long-step dual simplex will be used
+    40: mip =     not found yet >=              -inf        (1; 0)
+    40: >>>>>   0.000000000e+00 >=   0.000000000e+00   0.0% (1; 0)
+    40: mip =   0.000000000e+00 >=     tree is empty   0.0% (0; 1) INTEGER OPTIMAL SOLUTION
FOUND
GLPSOL--GLPK LP/MIP Solver 5.0
GLPK Integer Optimizer 5.0
1 hidden covering inequaliti(es) were detected
A: min|aij| =  1.000e+00  max|aij| =  1.000e+00  ratio =  1.000e+00
Problem data seem to be well scaled
Constructing initial basis...
Size of triangular part is 137
```

```
Solving LP relaxation...
GLPK Simplex Optimizer 5.0
209 rows, 192 columns, 820 non-zeros
      0: obj =   0.000000000e+00 inf =   3.000e+01 (27)
     62: obj =   0.000000000e+00 inf =   3.846e-02 (1)
LP HAS NO PRIMAL FEASIBLE SOLUTION
Time: 104 ms
```

PuLP klara alltså av att lös problem lpsolve inte kunde, däremot för det enkla sudokut var tiden i millisekunder högre med PuLP, vi tittar närmare på den data i kapitlet Jämförelser, Hastighet.

Backtracking i Python

Backtracking är en algoritm där man återgår till föregående steg eller lösning så snart man kan fastställa att den nuvarande lösning inte kan bli en komplett lösning. Vi kommer att använda denna princip för backtracking för att implementera följande algoritm.

Algoritmens uppgift är att leta efter tomma rutor i sudokun och sedan försöka placera 1–9 i rutan. När algoritmen har valt ut en ruta och placerat en siffra i den så kontrolleras det att siffran inte redan finns i y eller x axeln och inte i ett 3x3 rutnät. Ifall siffran inte är giltig går den tillbaka och försöker med nästa siffra 1-9. Ifall sifforna inte passar går den vidare till nästa ruta.

Detta upprepas för varje ruta tills det inte finns nå tomma rutor kvar. När det inte finns någon tom ruta kvar så har vi vår lösning.

Algoritmen klarar av att lösa alla sudokun vi valt ut.

Vi tar och löser samma sudokun som i LP-lösningarna:

```
solution 1 :
iterations: 213
Time:  1001000 NANO s
```

```
[[4 8 3 9 2 1 6 5 7]
 [9 6 7 3 4 5 8 2 1]
 [2 5 1 8 7 6 4 9 3]
 [5 4 8 1 3 2 9 7 6]
 [7 2 9 5 6 4 1 3 8]
 [1 3 6 7 9 8 2 4 5]
 [3 7 2 6 8 9 5 1 4]
 [8 1 4 2 5 3 7 6 9]
 [6 9 5 4 1 7 3 8 2]]
```

Löser lätta sudokun snabbare än vad PuLP gör.

```
solution 1 :
iterations: 427 579
Time:  640 ms
```

```
[[5 2 1 3 8 9 7 4 6]
 [4 3 8 7 6 5 9 1 2]
 [6 9 7 2 1 4 8 3 5]
 [3 6 5 8 9 1 2 7 4]
 [8 7 2 4 5 3 6 9 1]
 [1 4 9 6 7 2 3 5 8]
 [7 8 4 1 3 6 5 2 9]
 [2 5 6 9 4 7 1 8 3]
 [9 1 3 5 2 8 4 6 7]]
```

Backtracking tar längre tid att lösa svårare sudokun där förgreningsfaktorn är stor.

Genetisk algorithm i Python

En genetisk algoritm (GA) tar inspiration från naturens egen evolution. Utav en mängd olika lösningar, kallad population, två lösningar som har urvalts som goda korsas ihop för att skapa två nya "barn" vilka adderas till den nästa populationen. Många GA:n implementerar en chans åt barnens värden att muteras. Detta steg upprepas tills tillräckligt många barn har skapats, varefter den nya populationen blir evaluerad och korsad ihop. Nya populationer skapas ända tills en tillräckligt bra lösning har hittats eller mängden generationer överskrider programmets parametrar.

En GA består då av följande komponenter:

- En generator som skapar *n* mängd olika lösningar baserat på inputen
- En fitness evaluerare som rangordnar lösningarna
- Något som bestämmer vilka lösningar väljs för korsande
- En korsare för att kombinera två föräldrar till två barn
- En möjlighet för mutation
- Konditioner när GA ska sluta

Det sägs att GA är bra till att lösa TSP problem, vilket är huvudsakliga orsaken vi valt försöka lösa sudoku med GA är för att (enligt vår enkla uppfattning) sudoku kan ses som ett TSP där bara en nod("siffra") kan besökas ("placeras") utan att man besöker den noden igen (unik siffra per rad/kolumn/cell).

Och eftersom vi ser det som ett traveling salesman problem, använder vi PMX från tidigare hemuppgiften för att korsa lösningarnas x-axel rader. Under vår research hittade vi **inga andra artiklar med PMX**, vilket gör våra resultat och forskning unik!

Implementation

Input

Programmet läser in pussel given som en NumPy array. Baserat på pusslet genereras *n_parents* mängd slumpmässiga lösningar där de tomma platserna fylls med en permutation av 1-9. Siffrorna fylls i radvis så att inga konflikter uppkommer längs med x axeln, detta betyder dock att cellerna och kolumnvis kommer ha dubbletter. De för ifyllda siffrorna blir inte ändrade.

En större population ger större sannolikhet att GA:n konvergerar till en korrekt lösning, för en kostnad av minne och CPU-tid. Till våra easy-level sudokun tycks det räcka med en population av 3000.

Fitness

Fitness evalueraren är baserad på Fendrich 2010, där fitness är bestämt på hur många konflikter (dubbletter av siffror) lösningen har rad- kolumn- och cell vis. Lösningarna returneras sorterade med sina respektive poäng.

Val av lösningar

Den största utmaningen har varit hur man ska välja lösningar så att generationerna blir bättre, men inte fastna i lokala minimum med låg genetisk variation.

Vi började med att välja kandidater endast med sina vikter. Vikterna kalkylerades så att den bästa lösningen fick 1/2 vikt, den andrabästa 1/3, 1/4 osv. Detta ledde till populationer som kom till cirka 15 konflikter men konvergerade aldrig till en lösning, inte ens till otroligt lätta pussel som skulle lösas med bruteforce på någon sekund.

Till nästa lät vi para ihop lösningarna i rangordning och så att en lösning bara paras en gång. Detta ledde till en helt homogen population som inte kunde förbättras.

Degardin 2019 väljer de bästa 25% av populationen till att fortplanta med (en så kallad selection ratio).

Vi implementerade detta också och använder top 25% av lösningarna. Från de här 25% av lösningarna genereras *n_parents* antal nya barn. En fördel med selection ratio är försnabbad beräkning av en ny generation. Med detta lyckas GA:n lösa easy_02 vilket är en bra framgång, dock ännu en besvikelse då easy_02 har 31 ofyllda platser, varav många rader har bara 2 stycken ofyllda platser. De andra easy pusslena vill inte än konvergera.

Om ingen förbättring har skett de senaste 500 generationerna, antas det att GA:n har hamnat i ett lokalt minimum, så populationen startas om. Det vill säga den gamla generationen kastas helt ut och *generate_solutions()* skapar en ny slumpmässig population.

Vår sista ändring av denna komponent var att ta bort viktande helt. Generationernas diversitet blev högre och gjorde det möjligt att lösa alla easy level sudokun vi har.

Som sammanfattning, använder vi:

- Top 25% av generationens lösningar
- Slumpmässigt val av föräldrar
- Föräldrar kan paras ihop många gånger
- Ny population genereras om vi hamnar i ett lokalt minimum

Fortplantning och mutation

Vi har implementerat både PMX och slumpmässiga radbyten. Bara en av metoderna används åt gången.

PMX

Själva PMX:en är från tidigare hemuppgiften, bara modifierad att fungera åt andra moduler. PMX körs en rad i gången på slumpmässigt valda crossover punkter. Två versioner av barn genereras enligt PMX reglerna. De förinfyllda siffrorna används inte i crossovern.

Radbyten

En slumpmässig vald rad byts mellan föräldrarna. T.ex. rad x av förälder 1 och rad x + 1 av förälder 2.

Mutation

Om random.random() har ett värde mindre än mutation_rate muteras barnet. Två slumpmässiga platser på en slumpmässig rad byter plats på barnet. Vi fann att 10% mutation chans ökar på genetiska diversiteten utan att göra lösningarna o-konvergerande.

Resultat

Easy_01 - PMX

Parameter	Värde
Sudoku	easy_01
Korsare	PMX
Kromosomer	3000
Tid	248s
Generationer	1075
Restarts	1

Easy_01 - Row Swap

Parameter	Värde
Sudoku	easy_01
Korsare	Row Swap
Kromosomer	3000
Tid	595s
Generationer	4811
Restarts	4

PMX löser easy_01 med 1 generation restart, 248 sekunder i 1075 generationer.
Row swap löser easy_01 i 4 generation restarts, 595 sekunder och 4811 generationer.

PMX är dubbelt snabbare än row swap.

Easy_02 (supereasy) – PMX

Parameter	Värde
Korsare	PMX
Sudoku	easy_02(supereasy)
Kromosomer	3000
Tid	2.57s
Generationer	8
Restarts	0

Easy_02 - Row Swap

Parameter	Värde
Sudoku	easy_02(supereasy)
Korsare	Row Swap
Kromosomer	3000
Tid	7.29s
Generationer	53
Restarts	0

PMX lyckas lösa easy_02 i 2.57 sekunder på generation 8.
Row swap löser easy_02 i 7.29 sekunder på generation 53. PMX är bättre?!

Jämförelser

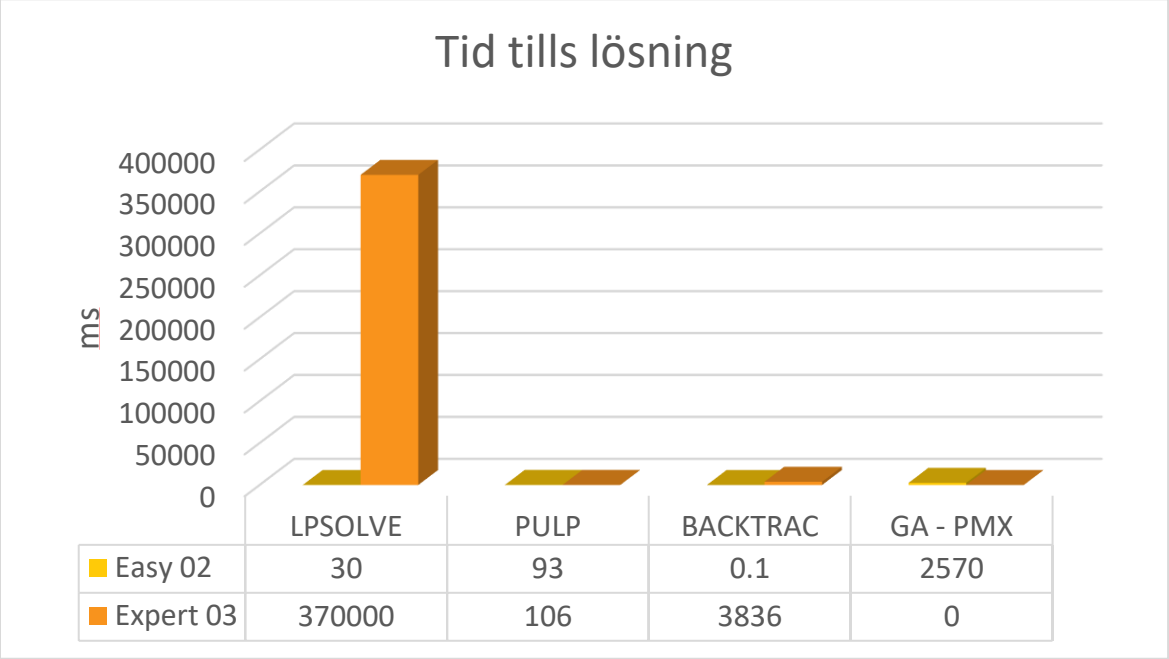
I avslutande kapitel ser vi lite övergripande på de olika metoder och algoritmer vi provat under projektets gång.

Lösningsförmåga

Vi kan enkelt konstatera att:

Lpsolve, klarar alla problem, iallafall i teorin, dock inte de svåraste inom en rimlig tid.
PuLP & Backtracking, klarar alla problem utan några större svårigheter
Genetiska Algoritmer, klarar en varierande mängde enklare problem i olika grad.

Hastighet



Med så stora skillnader i lösningstiderna blir vår grafik inte speciellt användbar och trots lpsolves stora stapel är det trots allt GA – PMX som bör intressera då den inte överhuvudtaget kunde lösa det svårare sudokut dock löser backtracking snabbast lätta puzzel men tar längre tid för att lösa en del av de svårare puzzlen. Hastigheten är mellan 1 ms och 4 sekunder. Eftersom algoritmen *n* kommer ihåg tidigare besökta celler så måste den göra beräkningarna på nytt. Detta gör att puzzel med mycket tomma rutor är tidskrävande att lösa.

Komplexitet

Utan att säga något kan det direkt göras en slutsats för att se hur kort och koncis backtracking koden är. Men vi skulle ändå argumentera att tankesättet bakom ett vanligt linjärt problem som lpsolve är ännu mindre komplext även om koden och lp-filen är rätt långa.

Medan inte något effektiv, att skapa en GA för att lösa sudokun är relativt simpelt och inte mycket hjärnaktivitet krävdes för att hitta på en metod som åtminstone har en chans att lösa en sudoku. Ungefär det enda som krävdes till extern källa var Fendrichs metod att evaluera en lösning (räkna ihop konflikter). Trots detta är metoden i sig både svårare att forma och mer komplex i vårt tycke än några av de andra.

Reflektioner

Som poängterat ett antal gånger, är GA inte en passande lösningsmetod för sudokupussel. Även så, har vi fyndat ett "nytt" sätt att lösa sudokun med PMX, och att PMX verkar lösa pusslen snabbare än row swap. Det skulle dock ännu vara intressant att veta ifall våran PMX skulle vara lika effektiv cell-vis istället för radvis, då det verkar många källor på nätet löser GA sudokun cell-för-cell istället.

Degardin 2019 till exempel löser cell-vis, och hen verkar kunna lösa betydligt svårare pussel än vår lösning. Vi har dock inte kollat igenom koden för att se vad de gör annorlunda utan får ta det en annan gång när tiden tillåter.

Vi har även sätt att det finns helt tydliga restriktioner med lpsolve och att mer robusta modeller som hittas i PuLP har en plats i optimering, eller kanske man bara borde hålla sig till mer ekonomiska problem istället för pussel...

Källor

Wikipedia, senast redigerad 2020, Sudoku, https://sv.wikipedia.org/, sedd 06 dec 2021, < https://sv.wikipedia.org/wiki/Sudoku>

Norvig, Peter, 2011, Solving Every Sudoku Puzzle, https://norvig.com/ , sedd 04 dec 2021, < https://norvig.com/sudoku.html >

Easybrain, senast redigerad 2021, https://easybrain.com/, sedd 04 dec 2021, < https://sudoku.com/ >

Computerphile, 2020, Nottingham University, Python Sudoku Solver, https://www.youtube.com/watch?v=G_UYXzGuqvM, sedd 06 dec 2021

Fendrich, D., 2010, Tillgänglig: http://fendrich.se/blog/2010/05/05/solving-sudoku-with-genetic-algorithms/ Hämtad: 9.12.2021

Dégardin, D., 2019, Tillgänglig: https://nidragedd.github.io/sudoku-genetics/ Hämtad: 9.12.2021