Examensarbete

HÖGSKOLAN I SKÖVDE

Tillämpbarheten av Learning Backtracking Search Optimization Algoritmen vid Lösning av Sudoku-problemet

The Application of the Learning Backtracking Search Optimization Algorithm when Applied to the Sudoku Problem

Examensarbete inom Informationsteknologi
Grundnivå 30 Högskolepoäng
Vårtermin 2017

Simon Sävhammar

Handledare: Jonas Gamalielsson
Examinator: Alan Said

Abstrakt

Den här rapporten undersöker egenskaper hos en algoritm som är baserad på *Learning Backtracking Search Optimization Algorithm* (LBSA) som introducerades av Chen et. al. (2017). Undersökningen genomfördes genom att tillämpa algoritmen på Sudokuproblemet och jämföra lösningsgraden och diversiteten i den sista populationen med en algoritm som är baserad på *Hybrid Genetic Algorithm* (HGA) som introducerades av Deng och Li (2011). Resultaten visar att implementationen av den LBSA-baserade algoritmen har en lägre lösningsgrad än den HGA-baserade algoritmen för alla genomförda experiment, men att algoritmen håller en högre diversitet i den sista populationen för tre av de fem gjorda experimenten. Slutsatsen är att den LBSA-baserade algoritmen inte är lämplig för att lösa Sudokuproblemet på grund av en låg lösningsgrad och att implementationen har en hög komplexitet.

Nyckelord: Evolutionär Algoritm, Sudoku, Optimeringsproblem.

Abstract

This report examines the properties of an algorithm based on the *Learning Backtracking Optimization Algorithm* (LBSA) introduced by Chen et. al. (2017). The examination was performed by applying the algorithm on the Sudoku problem and then comparing the solution rate and the diversity in the final population with an algorithm based on the *Hybrid Genetic Algorithm* introduced by Deng and Li (2011). The results show the implementation of the LBSA based algorithm have a lower solution rate than the HGA based algorithm for all executed experiments. But the LBSA based algorithm manage to keep a higher diversity in the final population in three of the five performed experiments. The conclusion is that the LBSA based algorithm is not suitable for solving the Sudoku problem since the algorithm has a lower solution rate and the implementation have a high complexity.

Keywords: Evolutionary Algorithms, Sudoku, Optimization Problems.

Innehållsförteckning

1 Introduktion....4
2 Bakgrund....4
2.1 Sudokupussel....4
2.2 Evolutionära algoritmer....5
2.2.1 Hybrida evolutionära algoritmer....6
2.2.2 Diversitet....7
2.2.3 TLBO....8
2.2.3.1 Lärarefasen....8
2.2.3.2 Inlärningsfasen....8
2.3 LBSA....8
2.3.1 Initiering av startpopulationerna....8
2.3.2 Selektion-I....9
2.3.3 Mutation....9
2.3.4 Crossover....10
2.3.5 Selektion-II....11
2.3.6 Korrigering av överskridande värden....11
2.4 HGA....11
3 Problemområde....12
3.1 Problemformulering och frågeställning....12
3.2 Delmål....13
4 Metod....13
4.1 Experiment....13
4.2 Implementation....14
5 Genomförande....14
5.1 Applikationsdetaljer....15
5.2 HGA-baserad jämförelsealgoritm....15
5.2.1 Selektionsstrategi....15
5.2.2 Crossover-operator....16
5.2.3 Mutationsoperator....17
5.2.4 Ersättningsstrategi....18
5.3 LBSA-baserad algoritm....18
5.3.1 Initiering av startpopulationerna....19
5.3.2 Selektionsstrategi....19
5.3.3 Mutationsoperator....19
5.3.4 Crossover-operator....20
5.3.5 Fylloperator....22
5.3.6 Ersättningsstrategi....23
6 Resultat och analys....23
6.1 Lösningsgrad....24
6.2 Diversitet....25
6.3 Svåra Sudokupussel....26
7 Diskussion....26
8 Validitetshot....28
9 Slutsats....28
Referenser....30
Appendix A Sudokupussel....32
Appendix B LBSA-baserad konstruktor....33
Appendix C LBSA-baserad Crossover-operator....34
Appendix D LBSA-baserad mutationsoperator....35
Appendix E LBSA-baserad sökning....37
Appendix F Randomiseringsfunktion....39

1 Introduktion

Målet med detta arbete är att tillämpa *Learning Backtracking Search Optimization Algorithm (*LBSA) (Chen, Zou, Lu och Wang, 2017) för att lösa Sudoku-problemet vilket kan utöka algoritmens applikationsområde.

BSA introducerades av Civicioglu (2013) för att lösa numeriska optimeringsproblem. Civicioglu (2013) kommer fram till att BSA kan lösa fler av de benchmarking-problemen som testades och att algoritmen kan vara mer effektiv än jämförelsealgoritmerna. LBSA är en ny algoritm som försöker förbättra algoritmen *Backtracking Search Optimization Algorithm* (BSA) genom att applicera *Teaching Learning Based Optimization* (TLBO) i mutationsfasen. Mutationsfasen är den fas där försökspopulationen genereras genom att använda information från individer i den nuvarande populationen. Målet med förändringen var att försöka förbättra det globala sökandet hos BSA. Chen et. al. (2017) anser viktiga framtida arbeten är att dra nytta av den enkla strukturen hos BSA genom att förbättra algoritmens förmåga att bibehålla diversiteten, med diversitet menas skillnaden mellan individerna i en population uppmätt med Hammingavståndet. Ett annat viktigt framtida arbete enligt Chen et. al. (2017) är utökandet av LBSAs applikationsområde.

Sudoku har enligt Aaronson (2006) lett till konkreta framsteg inom algoritmdesign och anses vara ett idealt problem för att undersöka klassen NP-kompletta problem. Enligt Segura et. al. (2016) har Sudoku varit ett brett använt problem för att testa egenskaperna hos nya optimeringstekniker.

Om LBSA visar sig lösa Sudokupussel effektivt finns det möjlighet att hitta flera praktiska applikationsområden eftersom många praktiska problem delar egenskaper med Sudokupussel. Det finns flera artiklar som undersöker hur ett problem kan lösas genom att göra om det till ett Sudokupussel eller som gör om ett Sudokupussel för att det ska likna ett praktisk problem. Exempelvis undersöker Okagbue et. al. (2015) hur Sudoku kan användas för att generera lösenord. Falcone (2013) använder en variant av Sudoku för att lösa ett schemaläggningsproblem för kirurger. Bufler et. al. (2017) använder Sudokupussel för att generera Sudokusekvenser vilka kan användas av en radardesigner.

Efter att rapporten introducerats i Kapitel Ett följer Kapitel Två som ger en beskrivning av viktiga begrepp som används i rapporten. Det tredje kapitlet presenterar problemområdet med forskning som gjorts tidigare och slutligen delmålen som ska undersökas. Det fjärde kapitlet presenterar de valda metoderna experiment och implementation. Kapitel Fem innehåller beskrivningen av implementationerna samt hur experimenten genomfördes. Kapitel Sex presenterar resultaten av experimenten samt analysen av data. Det sjunde kapitlet diskuterar resultaten av projektet. Kapitel Åtta behandlar validitetshot mot arbetet samt hur de är tänkta att hanteras. Kapitel Nio innehåller slutsatsen som arbetet med projektet ledde fram till. Bilagorna innehåller använda Sudokupussel, konstruktorn för implementationen av den LBSA-baserade algoritmen, operatorerna för den LBSA-baserade algoritmen, sökningsloopen för den LBSA-baserade algoritmen och randomiseringsfunktionen som används.

2 Bakgrund

2.1 Sudokupussel

Sudoku är ett NP-komplett numeriskt matrisbaserat pusselproblem där en $n^2 \times n^2$ matris har fyllts med siffror från ett till n på ett antal positioner (se figur 2.1.1). De från början givna siffrorna får inte byta plats under pusslets lösning, utan ska vara placerade på samma rutor i lösningen.

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

Figur 2.1.1: Exempel på Sudokupussel.

McGuire et. al. (2013) skriver att det inte finns något Sudokupussel som har 16 eller färre startsiffror och att ett korrekt Sudokupussel har exakt en lösning. Om det finns fler än en lösning har vissa rutor fler än en giltig siffra, lösaren måste gissa vilka siffror som ska vara i de rutorna, men meningen är att Sudoku ska vara ett logiskt pussel som kan lösas utan gissningar. Sudokupussel som har färre än 17 startsiffror har fler än en lösning och uppfyller därför inte villkoret att pusslet ska vara möjligt att lösa utan att lösaren bevöver gissa. Mcguire et. al. (2013) bevisade att det inte finns något Sudokupussel som har 16 eller färre startsiffror. Genom att katalogiserade de möjliga Sudokulösningarna och sedan applicera en *checker*-algoritm som de implementerat. *Checker*-algoritmen hittade inga Sudokupussel med 16 startsiffror som har en lösning men de hittade Sudokupusslet som har 16 startsiffror och 2 lösningar. När de testade *checker*-algoritmen på Sudokulösningar för att hitta Sudokupussel som har 17 startsiffror hittade de alla kända Sudokupussel. Lösningen på ett Sudoku pussel är en matris där varje rad, kolumn och $n \times n$ box har siffrorna ett till n exakt en gång. Enligt Yato och Seta (2003) är Sudoku ett NP-komplett problem.

Ytterligare problem som kan användas är för att undersöka området NP-kompletta problem är exempelvis Latinska rutor, den resande handelsmannen (eng. Travelling Sales Person), Masyu och Akari. Sudokuproblemet väljs på grund av argumenten i Aaronson (2006) och Segura et. al. (2016). En fördel med Sudoku framför andra möjliga problem är att det finns en optimal lösning och att det är enkelt att utvärdera när den är funnen. Det går att undersöka populationens diversitet vid en funnen lösning för att se om en jämförelsealgoritm har en bättre eller sämre diversitet än LBSA. Samt att det författaren anser det möjligt att anpassa LBSA till Sudokuproblemet men att det är svårare för Masyu-problemet och Akari-problemet. Latinska rutor hade varit ett möjligt val som delar flera av nämnda argument som Sudokuproblemet men som har färre restriktioner än Sudoku vilket gör att det är lättare att lösa. Författaren hittade heller inte några studier som löst problemet med hybrida evolutionära algoritmer, vilket gör att det inte finns någon jämförelsealgoritm.

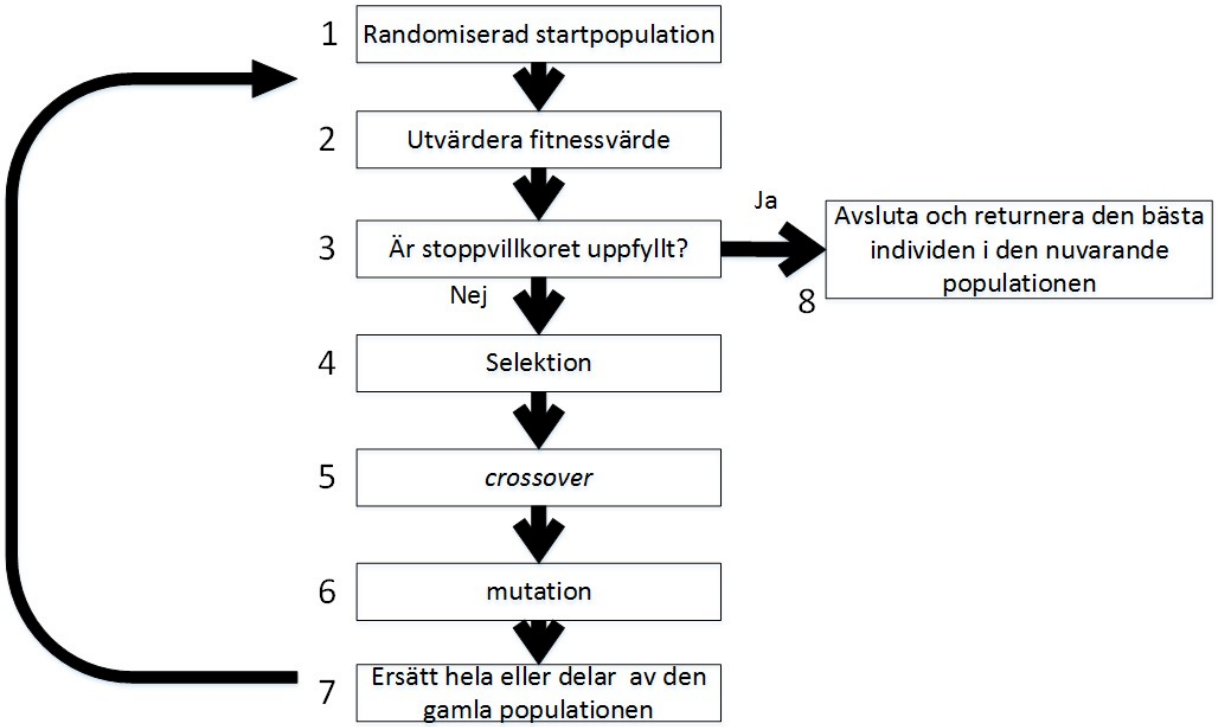
2.2 Evolutionära algoritmer

Evolutionära algoritmer (EA) är en kategori av algoritmer som bygger populationsbaserad stokastisk sökning (Civicioglu, 2013). Redan Alan Turing introducerade idén med evolutionära processer i sina skrifter 1948-1950 (Brabazon, O'Neill och McGarraghy, 2015) Under 1960-talet och 1970-talet presenterades viktiga framsteg inom evolutionära algoritmer samtidigt i USA och Tyskland. Gemensamt för evolutionära processer är enligt Brabazon, O'Neill och McGarraghy (2015) att de har en population av individer, selektion baserad på individernas fitnessvärde bibehållandet av individer med bra fitnessvärde och en generation med nya individer som ersätter hela eller delar av den gamla populationen. Fitnessvärdet är ett värde som används för att bedöma hur nära lösningen en individ är. Brabazon, O'Neill och McGarraghy (2015) skriver att även om forskningen om de evolutionära metoderna som nämnts tidigare förut var distinkt, har skillnaderna blivit mindre och idag är det vanligt att använda begreppet evolutionära algoritmer för flera evolutionära metoder.

Genetiska algoritmer (GA) har sedan 1960-talet använts för att lösa optimeringsproblem. Preux och Talbi (2009) skriver att enligt deras erfarenheter är GA väl anpassade för att lösa optimeringsproblem i sökområden med flera dimensioner.

Figur 2.2.1 visar ett flödesschema över hur en generell GA kan designas. En GA måste inte utföra både en crossover-operation och en mutationsoperation, men det är vanligt att båda operationerna är inkluderade. Crossover-operationen använder information från två individer för att generera en eller flera nya individer,

under mutationsoperationen förändras en individ slumpmässigt för att individer inte ska bli för lika varandra.



Figur 2.2.1: Flödesschema över en generaliserad GA.

I det första steget skapas en startpopulation med randomiserade individer, i det andra steget appliceras målsättningsfunktionen som utvärderar hur nära en individ är den optimala lösningen. Målsättningsfunktionen måste designas för varje GA och returnerar ett fitnessvärde, i vissa fall kan värdet som returneras behövas normaliseras innan det kan användas som fitnessvärde. I det tredje steget utvärderas om stoppvillkoret är uppfyllt, är det uppfyllt avslutas sökningen (8), om inte går den vidare. Exempel på stoppvillkor är en individ med ett fitnessvärde över eller under ett visst värde har hittats eller ett förutbestämt antal iterationer har genomförts. I det fjärde steget, selektion, väljs ett antal individer från populationen för att i nästa steg generera avkomma. Selektionen kan utföras på ett antal olika sätt men det är vanligt att individernas fitnessvärde ligger till grund för sannolikheten att väljas. I det femte steget *crossover* generas nya individer genom att kopiera information från två individer (föräldrar) som valts under selektionssteget, den nya individen har information från båda föräldrarna. Genom *crossover*-operatorn ärvs flera gener från två individer till avkomman för att styra sökningen till individer som representerar bra lösningar (Brabazon, O'Neill och McGarraghy, 2015). I det sjätte steget, mutation, muteras individer för att förhindra att individer blir för lika varandra och att sökningen därmed stannar. Om mutation inte utförs finns en risk att *crossover*-operatorn producerar avkomma som är identisk med redan existerande individer, sökningen kan därmed inte utforska nya sökområden och kan ha fastnat i ett lokalt optimum. Mutationsoperatorn måste balanseras för att den inte ska förstöra individer med bra fitness innan de har hunnits exploaterats av selektionen och crossover-operatorn. Ju mer selektionen och *crossover*-operatorn exploaterar bra individer, desto större behöver sannolikheten vara att individer muteras (Brabazon, O'Neill och McGarraghy, 2015). I det sjunde steget sker ersättningen av den gamla populationen. Det finns flera olika ersättningsstrategier som kan användas för att ersätta den gamla populationen. En variant är att delar av populationen ersätts av nyligen genererad avkomma, om delar av populationen ersätts brukar individer med bra fitnessvärde behållas medan individer med dåligt fitnessvärde kasseras. En annan variant är att hela den gamla populationen ersätts av den nya generationen, i vilket fall fitnessvärdet inte används under ersättningen.

2.2.1 Hybrida evolutionära algoritmer

Hybrida evolutionära algoritmer är en kombination av en evolutionär algoritm och andra optimiseringstekniker, heuristik eller maskininlärningstekniker. Cotta och Troya (1998) definierar

evolutionära metoder som använder arkitekturen från en EA kombinerat med kunskap om problemdomänen, som hybrida evolutionära algoritmer. De skriver även att hybridisering kan åstadkommas genom att antingen förändra algoritmens inre komponenter baserat på kunskapen om problemdomänen eller genom att kombinera flera sökalgoritmer med varandra.

2.2.2 Diversitet

Det finns olika sätt att definiera och mäta diversitet på, och de valda definitionerna och måtten kan ge olika resultat för samma populationer. Hammingavståndet är en vanlig teknik för att mäta skillnaden mellan genotyper (Morrison och De Jong, 2002) och har en låg komplexitet O(n). Hammingavståndet är lika med antalet positioner som är olika mellan två lika långa strängar. Entropi ger en bättre bild av individernas spridning (Črepinšek et. al., 2013) men författaren var inte säker på att måttet kunde implementeras korrekt.

Diversiteten kan mätas på fenotyper och genotyper. Diversitet på fenotyper definieras av Črepinšek et. al. (2013) som skillnaden mellan fitnessvärden inom en population, medans diversitet på genotyper definieras som skillnaden mellan individer i en population. Ett exempel på att mäta diversitet på fenotyper är att kategorisera individer efter deras fitnessvärde och sedan använda antalet kategorier som mått på diversiteten. Enligt Smit et. al. (2011) ger mätningar på genotyp-nivå bättre information om individernas spridning, på grund av att ett oändligt antal individer kan vara kopplade till ett fitnessvärde vid mätning på fenotyp-nivå. I detta arbete mäts diversiteten (se nedanstående beskrivning av diversitet) på genotyper därför att individer med samma fitnessvärde kan ha stora skillnader i arrayerna som representerar Sudokupusslet. För att mäta diversiteten på genotyp-nivå finns flera möjliga tekniker, Črepinšek et. al. (2013) kategoriserar dem i fem klasser, differens, avstånd, entropi, sannolikhet och släktskap. Exempel på mått från differens är antalet unika genotyper och aktiverade noder där en nod representerar ett område i sökområdet. Ett exempel på ett mått från släktskap är antalet individer i den första populationen som bidragit med information till individer i den sista populationen. Att räkna antalet unika genotyper kommer leda till att algoritmerna får liknande diversitet, eftersom det för troligen bara är individer med låga fitnessvärden som blir identiska. Att mäta unika genotyper gör ingen skillnad på hur mycket skillnad det är mellan olika individer. Om två individer har flera positioner som är olika bör det påverka diversiteten mer än om två individer har få positioner som är olika. Släktskap är svårt att implementera eftersom antalet giltiga symboler i en ruta är begränsat, endast nio möjliga. Det blir svårt att avgöra om en siffra kommer från en individ i den första positionen eller om den är skriven under en senare iteration. Entropi hade varit ett möjligt mått att mäta diversitet och kan ge en bättre bild av diversiteten än mått från avståndsklassen, men det är också komplex och tidskrävande att implementera. Därför valdes klassen avstånd för att mäta antal positioner som skiljer sig mellan två individer.

En individ representerar ett ifyllt Sudokupussel och diversitet i detta arbete tolkas som antalet positioner som skiljer sig mellan Sudokupussel. Där en viss position refererar till en ruta i en bestämd rad, kolumn och *n x n*-box. Exempelvis kan position tre referera till den i figur 2.2.2 markerade rutan, som ligger i den första raden, den tredje kolumnen och *n x n*-boxen högst upp till vänster.

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

Figur 2.2.2: Sudokupussel med position tre markerad.

Den valda representationen av en individ är ett Sudokupussel är en array med 81 element och alla individer har lika långa arrayer. För att mäta skillnaden mellan arrayer finns flera möjliga mått exempelvis Levenshteinavståndet. Levenshteinavståndet mäter hur många karaktärer som måste läggas till, ersättas och raderas, för att en sträng ska bli lik en annan sträng och kräver inte att strängarna är lika långa.

Levenshteinavståndet kommer ge ett liknande resultat som Hammingavståndet, men ger i vissa fall ett lägre värde än Hammingavståndet, exempelvis om två strängar kan bli identiska genom att ta bort den första karaktären och lägga till en karaktär på slutet av strängen.

2.2.3 TLBO

TLBO är en optimeringsalgoritm som introducerades av Rao et. al. (2011) och som inspireras av hur inlärning fungerar i ett klassrum. Algoritmen har två inlärningsfaser, lärarefasen och inlärningsfasen.

2.2.3.1 Lärarefasen

Under lärarefasen lär sig lärande individer av en lärare och medelvärdet för ett ämne, ett ämne kan exempelvis vara en av de variablerna som ska optimeras.

$$X_{\text{new, i}} = X_{\text{old, i}} + rand * (\mathrm{X}_{Teacher} - \mathrm{T}_F \mathrm{X}_{Mean})$$ *Ekvation 2.2.1*

Ekvation 2.2.1 visar ekvationen för lärarefasen. Där $X_{new,i}$ är den nya positionen som genereras av ekvationen, *i* identifierar den lärandeindividen. $X_{old,i}$ är den gamla positionen för *i,* $X_{Teacher}$ den bästa individen för ämnet i populationen, T_F är lärarfaktorn som slumpas med lika stor sannolikhet till ett eller två, och X_{Mean} är medelvärdet för ämnet.

För att hitta det optimala värdet för T_F utfördes enligt Rao (2016) experiment på benchmarking-funktioner. Resultaten av de experiment var att algoritmen presterade bäst om värdet för T_F slumpades till ett eller två.

2.2.3.2 Inlärningsfasen

I lärandefasen lär sig varje individ i populationen från en annan individ som välj slumpvis, uppdateringen av individens position utförs enligt en av två ekvationer.

$$X_{\text{new, i}} = X_{\text{old, i}} + rand * (X_{\text{old, i}} - X_{\text{old, k}})$$ *Ekvation 2.2.2*

Valet av ekvation beror på om positionen för den slumpade individen har ett lägre eller högre värde än positionen för individ *i.* För ett minimeringsproblem gäller att om värdet för den slumpade individen är lägre används ekvation 2.2.2, För ett maximeringsproblem gäller att om värdet för den slumpade individen är lägre används ekvation 2.2.3.

$$X_{\text{new, i}} = X_{\text{old, i}} + rand * (X_{\text{old, k}} - X_{\text{old, i}})$$ *Ekvation 2.2.3*

Om värdet för den slumpade individen är högre än värdet för individ *i* används ekvation 2.2.3 för ett minimeringsproblem. För ett maximeringsproblem används ekvation 2.2.2. TLBO använder girig selektion vilket innebär att bara den bästa positionen av X_{new} och X_{old} sparas både under lärarefasen och under inlärningsfasen.

2.3 LBSA

LBSA introducerades av Chen et al. (2017) för att förbättra egenskaper hos BSA (Civicioglu, 2013). LBSA är en evolutionär algoritm för lösning av numeriska optimiseringsproblem där numren i problemet är reella tal. Både BSA och LBSA har ett minne i form av en historisk population som används under mutationsoperationen för att uppdatera individer i försökspopulationen som innehåller de nya individerna som generas under en iteration. I beskrivningen av LBSA används betäckningen *rand* för en funktion som returnerar ett reellt normaldistribuerat värde mellan noll och ett.

2.3.1 Initiering av startpopulationerna

Under initieringsfasen sätts startvärde för alla individer i startpopulationen och den historiska populationen.

```
for i <= population size

       for j <= D

              Pi,j ← lowj + rand * (upj - lowj)

              oldPi,j ← lowj + rand * (upj - lowj)

       endfor

endfor

Evaluate all individuals

memorize the best and the worst individual
```

Algoritm 2.3.1: Initialisering av start- och historiskpopulation.

Algoritm 2.3.1 beskriver initieringen av startpopulationen *P* och den historiska populationen *oldP*. Populationerna består av *i* individer och varje individ har en array med flyttal, arrayen har storlek *D*. Det lägsta värdet som kan sättas för en individ är low_j, det högsta värdet är up_j, *rand* slumpar ett reellt tal mellan noll och ett.

2.3.2 Selektion-I

Selektion-I utförs på samma sätt som i BSA om ett villkor uppfylls ersätter den nuvarande populationen den historiska populationen. Om den historiska populationen ersätts blandas individerna.

```
if (a < b  | a,b ~U(0, 1))

     oldP ← P

     shuffle(oldP)

endif
```

Algoritm 2.3.2: Ersättning av den historiska populationen.

Algoritm 2.3.2 innehåller pseudokod för ersättningen av den historiska populationen i LBSA.

Om villkoret $a < b$ uppfylls ersätter den nuvarande populationen *P* den historiska populationen *oldP*. Under den första iterationen innehåller *P* individerna som generades och initierades under initieringsfasen, under följande iterationerna innehåller *P* de individer som överlevde selektion-II.

2.3.3 Mutation

LBSA kombinerar TLBO och BSA för att öka sammanstrålningshastigheten genom att guida inlärningen under mutationsoperationen. LBSA har två mutationsekvationer men en individ muteras bara av den ena, valet av ekvation slumpas. Under mutationen genereras lika många nya individer som finns i startpopulationen, de nya individerna lagras i försökspopulationen.

För den första mutationsekvationen som används av LBSA, utökas mutationen från BSA med information från individen med lägst fitnessvärde i den nuvarande populationen.

$$Mutant = P + F * (oldP - P)$$

Ekvation 2.3.1

I BSA används information från den historiska populationen *oldP* och den nuvarande populationen *P* för att generera en ny position under mutationen (ekvation 2.3.1), enligt Civicioglu (2013) används *F* för att kontrollera hur mycket (*oldP – P*) påverkar sökningen. Värdet för *F* sätts till 3 * *rand* i både Civicioglu (2013) och Chen et. al. (2017).

$$M = P + F * (0.5 * (oldP - P) + 0.5 * rand * (Teacher - P))$$ *Ekvation 2.3.2*

Ekvation 2.3.2 är den första mutationsekvationen som används av LBSA. Där *M* är försökspopulationen, *P* är den nuvarande populationen, *F* är en konstant som sätts till 3 * *rand*, o*ldP* är den historiska populationen och *Teacher* är individen i den nuvarande populationen med det bästa fitnessvärdet.

För den andra mutationsekvationen använder LBSA lärarefasen och inlärningsfasen från TLBO. LBSA modifierar både lärarefasen inlärningen från TLBO genom att göra om den till tre steg och sedan kombinera de tre stegen till en ekvation (ekvation 2.3.3). I det första steget lär sig en individ från en annan slumpad individ, under det andra steget från den bästa individen i population (*Teacher)* och i det sista steget från den sämsta individen i populationen (*Worst*).

$$\begin{gathered} \textit{om individ i är bättre än individ k} \\ M_{i,j} = P_{i,j} + rand * (P_{i,j} - P_{k,j}) + rand * (Teacher_j - P_{i,j}) + rand * (Worst_j - P_{i,j}) \\ \textit{om individ k är bättre än individ i} \\ M_{i,j} = P_{i,j} + rand * (P_{k,j} - P_{i,j}) + rand * (Teacher_j - P_{i,j}) + rand * (Worst_j - P_{i,j}) \end{gathered}$$ *Ekvation 2.3.3*

M är en individ i försökspopulationen, *P* är en individ i den nuvarande populationen, *Teacher* är den individen i den nuvarande populationen som har bästa fitnessvärdet, och *Worst* är den individ i den nuvarande populationen med det sämsta fitnessvärdet.

2.3.4 Crossover

LBSAs *Crossover*-operator (algoritm 2.3.3) fungerar på samma sätt som i BSA. En matris med storleken *pSize x pDim* fylls med ettor. Där *pSize* är storleken på populationen och *pDim* är längden på arrayen men de flyttal som ska optimeras. Sedan väljs en av två varianter för att bestämma hur många positioner som ska kopieras från en individ i populationen till en individ i försökspopulationen.

```
crossoverMap[pSize][pDim] fill all positions with 1

if (a < b  | a,b ~U(0, 1))

       for i <= pSize

            crossoverMap_{i,u:ceil(mixrate * rand *D)} ← 0  | u = permute(1, 2, 3, …, D)

       endfor

else
       for i <= pSize

              crossoverMap_{i,rand(D)} ← 0

endif

T ← Mutant

for i <= pSize

       for j <= pDim

              if crossoverMap_{i,j} == 1

                     T_{i,j} ← P_{i,j}

              endif

       endfor

endfor
```

Algoritm 2.3.3: LBSAs Crossover-operator.

Den första varianten itererar över matrisen *crossoverMap* och sätter ett antal slumpade positioner till noll, *u* är arrayen för problemdimensionen *D*, *mixrate* är en variabel som bestämmer hur många positioner av problemdimensionen som maximalt kan sättas till noll. Randomiseringen av positionerna fås genom att arrayen som innehåller positionernas index blandas. Den andra varianten sätter en position i arrayen för en individs problemdimension till noll. När en av varianterna har utförs sätts *T* till försökspopulationen som skapats under mutationsoperatorn, sedan itererar en nästlad loop över matrisen som skapats i början av funktionen. Den yttre loopen itererar över arrayen som representerar alla individer i populationen och den inre över arrayen som representerar problemdimensionen. När den inre loopen hittar en position som innehåller värdet ett, kopieras värdet som finns på den positionen i arrayen för problemdimensionen för den aktuella individen i populationen. Till arrayen för problemdimensionen för motsvarande individ i försökspopulationen. När funktionen har kört färdigt har alla individer i försökspopulationen korsats med motsvarande individ i populationen.

2.3.5 Selektion-II

LBSAs ersättning av individer i populationen är helt elitistisk. Om en individ i försökspopulationen har ett fitnessvärde som är bättre eller lika med fitnessvärdet för individen på samma position i populationen. Ersätter individen i försökspopulationen den individen (algoritm 2.3.4).

```
for i < population size

     if Ti.fitness <= Pi.fitness

          Pi ←  Ti

     endif

endfor
```

Algoritm 2.3.4: Ersättning av individer i populationen med individer i försökspopulationen.

Där T_i är individ *i* i försökspopulationen och P_i är individ *i* i den nuvarande populationen.

2.3.6 Korrigering av överskridande värden

LBSA designades för att optimera arrayer med reella tal, där variablerna i arrayen har ett lägsta och ett högsta värde. Därför utförs en korrigering av värden som ligger utanför det tillåtna intervallet vilket finns beskrivet i Chen et al. (2017).

2.4 HGA

HGA som introducerades av Deng och Li (2011) är en algoritm som har enkel implementation men ändå har en hög lösningsgrad för lätta och medelsvåra Sudokupussel. För lätta Sudokupussel är lösningsgraden mellan 80 till 100 procent, med ett medelvärde för antal iterationer mellan 200 och 500. För medelsvåra Sudokupussel är lösningsgraden 60 procent och medelvärdet för antal iterationer runt 600. För svåra Sudokupussel är lösningsgraden endast sjutton procent. Det är svårt att jämföra algoritmer mellan författare eftersom de Sudokupussel som lösts inte alltid är angivna i rapporten, exempelvis anger Deng och Li (2011) från vilken sida de hämtat pussel men inte vilka Sudokupussel som använts. Om lösningsgraden för HGA jämförs med exempelvis Chel et. al. (2016) ser det ut som HGA har en bättre lösningsgrad för alla lätta, medelsvåra och svåra Sudokupussel, men det är svårt att avgöra om Deng och Li (2011) och Chel et. al. (2016) använder pussel som är jämförbara. En algoritm som överlag har en bättre lösningsgrad än HGA är Wang et. al. (2015) för lätta och medelsvåra pussel är lösningsgraden 100 procent, för svåra en lösningsgrad mellan 96 och 100 procent och för supersvåra en lösningsgrad mellan 58 och 98 procent. Ett försök att implementera Wang et. al. (2015) gjordes men den algoritmen lyckades inte lösa Sudokupussel, och eftersom ingen pseudokod publicerats var det svårt att identifiera vad som var fel. HGA valdes för att den lyckas lösa flera lätta och medelsvåra Sudokupussel, samtidigt som den har en enkel implementation.

3 Problemområde

NP-kompletta matrisbaserade pusselproblem är problemen som består av en *n x n* matris som delvis har fyllts med ett antal symboler, målet är att hitta en lösning som uppfyller vissa villkor exempelvis att de tomma rutorna ska fyllas med de symboler som saknas (Latinska rutor), att rader och kolumner ska ha en viss summa (Kakuro) eller att linjer ska ritas mellan likadan symboler (Zig-zag numberlink).

Sudoku är ett problem under området kombinatorisk spelteori, exempel på andra arbeten inom området är Adcock et al., (2015) som undersöker om ett specifikt problemet är NP-komplett och Wang et. al. (2015) som undersöker effektiviteten hos en föreslagen algoritm.

Forskare har undersökt effektiviteten hos flera olika tekniker för att lösa Sudoku, exempelvis Yue och Lee (2006) som beskriver en teknik för att implementera ett kvantum-neuronnät för att lösa Sudokupussel. Jilg (2009) jämför i sin rapport sex populationsbaserad algoritmer med varandra, Kamal, Chawla och Goel (2015) jämför den relativa effektiviteten hos *Backtracking, Simulated Annealing,* och genetiska algoritmer.

Från 2006 och framåt har det publicerats artiklar där forskare undersöker hur effektiva evolutionära algoritmer är på att lösa Sudoku, en av de tidigaste rapporterna undersöker om en genetisk algoritm som använder grammatisk evolution kan lösa Sudoku, (Nicolau och Ryan, 2006) och deras resultat visar att den klarar att lösa problemet, men effektiviteten hos den teknik de använde sätts inte i relation till andra existerande tekniker.

Mantere och Koljonen (2007) undersökte om det går att lösa Sudokupussel med genetiska algoritmer och kom fram till att det är möjligt men att det finns mer effektiva tekniker än den genetiska algoritmen de presenterade. De föreslår att framtida forskning ska undersöka om hybrida genetiska algoritmer är mer effektiva på att lösa Sudokupussel. Asif och Baig (2009) experimenterar med *Ant Colony Optimisation,* deras algoritm klarar att fylla upp till 76 av 81 rutor i polynomiell tid men klarar inte att lösa alla rutor.

På senare år har forskare försökt förbättra effektiviteten hos evolutionära algoritmer genom att kombinera dem med andra algoritmer. Exempelvis föreslår Mantere och Koljonen (2009) en genetisk algoritm som kombineras med en *Ant Colony Optimization* algoritm. Men de undersöker inte hur diversiteten i populationen påverkar algoritmens förmåga att lösa problemet. Chen et al. (2017) presenterar en ny evolutionär algoritm LBSA som kombinerar två algoritmer som framgångsrikt har löst optimiseringsproblem, BSA och TLBO. Enligt Chen et al. (2017) har LBSA en snabbare konvergeringshastighet än BSA samtidigt som den bibehåller en diversitet i populationen.

3.1 Problemformulering och frågeställning

Hybrida evolutionära algoritmer har varit relativt framgångsrika på att lösa Sudoku (Chel, Mylavarapu & Sharma, 2016) och det finns ett intresse av att utöka applikationsområdet för BSA och LBSA (Chen et al., 2017). Enligt Civicoglu (2013) jämfördes BSA med flera andra evolutionära algoritmer som ofta används i litteratur. Resultaten av studien indikerar att BSA generellt är bättre på att lösa flera av problemen som testades, speciellt på de mer komplexa problemen lyckades BSA bättre än jämförelsealgoritmerna. Ytterligare analyser visade att BSA löste problemen snabbare än flera av de andra algoritmerna. LBSA försöker förbättra mutationsoperatorn i BSA, bland annat genom att förbättra populationens diversitet. Segura et. al. (2016) diskuterar vikten av diversitet vid lösning av Sudokupussel. Frågan är om förbättringarna av BSA som Chen et. al. (2017) utför gör algoritmen tillämpbar på Sudokuproblemet?

Om LBSA effektivt kan lösa NP-kompletta matrisbaserade pusselproblem finns ett brett applikationsområde för algoritmen. Om den samtidigt bibehåller diversiteten i populationen kan den vara bättre på att lösa problem som har fler än en lösning än tidigare algoritmer. Exempelvis kan schemaläggningsproblem och Sudokupussel ha fler än en lösning.

Forskningen som författaren har tagit del av har inte undersökt om diversiteten hos en population kan förbättra effektiviteten hos en evolutionär algoritm vid lösning av Sudokupussel. Denna rapport undersöker därför frågan om LBSA effektivt kan lösa NP-kompletta matrisbaserade pusselproblem genom att applicera algoritmen på Sudokupussel och om algoritmen bibehåller diversiteten i populationen. Effektiviteten undersöks genom att jämföra LBSA med en hybrid evolutionär algoritm som kallas *Novel Hybrid Genetic Algorithm* (HGA), algoritmen introducerades av Deng och Li (2011). Om diversiteten i populationen

bibehålls undersöks genom att mäta avståndet mellan individer när en lösning har hittats.

3.2 Delmål

Det första delmålet undersöker om LBSA effektivt kan lösa Sudokupussel genom att jämföra med HGA. Värdet som jämförs är antalet generationer varje algoritm behöver för att hitta en lösning. Eftersom LBSA har en förbättrad diversitet än den tidigare BSA är det möjligt att det påverkar algoritmens konvergeringshastighet negativt, det kan betyda att HGA snabbare hittar lösningen på speciellt enklare Sudokupussel. Meningen med delmål ett är därför att besvara frågan om LBSA kan realiseras som en Sudoku-lösningsalgoritm även om konvergeringshastigheten har försämrats. Om det är möjligt det troligt att algoritmen är användbar för att lösa andra NP-kompletta problem.

1. *Fastställa hur LBSA och HGA skiljer sig med avseende på antal generation för att lösa Sudokupussel*

Det andra delmålet prövar om LBSA bibehåller mer diversitet i populationen än HGA vid lösning av Sudokupussel. Enligt Chen et al. (2017) har de förbättrat sökningsförmågan hos den existerande BSA genom att introducera *Teaching-Learning* element för att öka diversiteten hos individerna i populationen. Delmål två undersöker därför om LBSA håller mer diversitet i populationen än HGA och därmed har en bredare sökningsförmåga, vilket kan vara till nytta för att lokalisera fler än en lösning på Sudokupussel.

2. *Undersöka om LBSA är bättre på att behålla diversitet i populationen än HGA, när individerna i en population representerar ett Sudokubräde fyllt med siffror mellan ett och nio.*

Det tredje delmålet är att kontrollera om LBSA mer frekvent än HGA kan lokalisera lösningen på svåra Sudokupussel som har få givna siffror. HGA får enligt Deng och Li (2011) problem med svåra Sudokupussel, det är möjligt att LBSA kan vara bättre på att lokalisera en lösning på de svåra problemen eftersom algoritmen inte omedelbart konvergerar mot en lovande individ.

3. *Undersöka om LBSA kan lösa fler av de svåra Sudokupusslen än HGA.*

Förväntat resultat av alla delmål är statistiskt säkerställd data som visar hur många generationer LBSA och HGA behöver för att lösa Sudoku pussel. Resultatet presenteras med grafer där båda algoritmerna är representerade. Exempelvis antalet generationer presenteras på y-axeln och tiden för experimentet presenteras på x-axeln. Statistiska tester används för att avgöra om en sampling av mätvärden tenderar att innehålla mindre värden än en annan sampling.

4 Metod

För att avgöra om LBSA är lämplig att använda föra att lösa Sudoku pussel och om den kan lösa Sudoku med ett mindre antal generationer än HGA har metoderna implementation och experiment valts. Metoden implementation används för att det ska vara möjligt att realisera LBSA och HGA på ett Sudoku pussel, och metoden experiment gör det möjligt att jämföra antalet generationer och att skillnaden är statistiskt säkerställd.

4.1 Experiment

Denna rapport undersöker frågeställningen huruvida LBSA är en effektiv algoritm för att lösa Sudokupussel genom tre delmål. Eftersom målet är att bekräfta eller falsifiera en hypotes med ett fåtal variabler, antalet generationer och diversiteten i en population, anses experiment vara en lämplig metod för att besvara frågeställningen. Enligt Wohlin (2012) är experiment lämpligt för att undersöka frågeställningar som till exempel att bekräfta teorier, undersöka relationer, utvärdera modeller och för att kontrollera att mätningar är korrekta. Berndtsson et al. (2008) hävdar att experiment undersöker ett fåtal variabler, hur de påverkas av förhållanden i experimentet och att experiment används för att bekräfta eller falsifiera hypoteser. Experiment är även en vanlig metod i rapporter som undersöker liknande frågeställningar exempelvis används den i en artikel som undersöker effektiviteten hos BSA genom att utföra tester och jämföra resultaten med andra algoritmer (Civicioglu, 2013).

4.2 Implementation

Implementation kan enligt Berndtsson et al. (2008) vara en lämplig metod för att implementera en ny lösning på ett problem inom datavetenskap och informationssystem. Metoden innebär att en mjukvaruarkitektur, procedur, algoritm eller teknik implementeras för att visa att den föreslagna lösningen antingen beter sig på ett visst sätt eller innehar vissa egenskaper. För att undvika validitetshot och reliabilitetshot är det viktigt att implementationen görs enligt god mjukvaruutvecklingssed. För att det ska vara möjligt att utföra de experiment som undersöker om LBSA är en effektiv algoritm för att lösa Sudokupussel måste det finnas en implementation av algoritmen. Det har inte varit möjligt att få tillgång till källkoden från forskarna som föreslagit LBSA (Chen et al., 2017) eller HGA (Deng och Li, 2011), därför måste nya versioner implementeras. De nya versionerna görs från den information som finns tillgänglig i de publicerade artiklarna. Men eftersom det inte går att kontrollera om de är korrekt implementerade inkluderas källkoden för de viktiga delarna av algoritmerna i appendix delen av denna rapport.

5 Genomförande

För att uppnå delmålen kommer den vetenskapliga metoden experiment att användas för att bekräfta eller falsifiera hypoteser, hypoteserna formuleras baserat på delmålen.

Hypotes 1:

LBSA kan lösa Sudokupussel på ett mindre antal generationer än HGA.

Hypotes 2:

LBSA bibehåller en större diversitet i populationen än HGA.

Hypotes 3:

LBSA hittar mer frekvent det globala optimat än HGA.

Tolkningar av LBSA och HGA implementeras i programmeringsspråket C++, tolkningarna baseras på den information som finns tillgänglig i forskningsartiklarna. LBSA är beskriven i Chen et al (2017), och HGA beskrivs i Deng och Li (2011).

Rapporterna innehåller beskrivningar av algoritmerna och vissa delar finns förklarade med pseudokod Algoritmerna beräknas därför vara möjliga att implementera på ett sätt som gör det möjligt att testa deras effektivitet vid lösning av Sudokupussel.

LBSA och HGA är evolutionära algoritmer och generar vid sökningens början ett antal individer som ingår i en population där varje individ har ett fitnessvärde som beräknas av en fitnessfunktion.

Målsättningsfunktionen är en viktig del av evolutionära algoritmer som använder fitnessvärde och kan ha stor påverkan på en algoritms effektivitet. Därför används en identisk målsättningsfunktion för både den LBSA-baserade och den HGA-baserade algoritmen, eftersom det är andra delar av algoritmerna som är intressanta att utvärdera, exempelvis hur effektiv *Teaching-Learning* tekniken som LBSA applicerar i mutationsfasen är för att bibehålla diversiteten i populationen. Den valda målsättningsfunktionen (ekvation 5.1) används av Deng och Li (2011) i artikeln som beskriver HGA, och räknar antalet återkommande nummer i varje rad och kolumn i Sudokupusslet.

$$w = \sum_{i=1}^{9} (r_i + c_i) \qquad \textit{Ekvation 5.1}$$

Där r_i är antalet återkommande nummer rad i och c_i är antalet återkommande nummer i kolumn i, när fitnessvärdet är noll har algoritmen hittat en lösning på Sudokupusslet.

En testmiljö är implementerad i C++ som kan läsa in ett Sudokupussel och applicera den LBSA-baserade eller den HGA-baserade algoritmen för att söka efter en lösning. Experimenten kommer att utföras genom att ett Sudokupussel läses in i testmiljön, och en av algoritmerna användes därefter för att söka efter en lösning.

Experimentet som testar hypotes ett applicerar den LBSA-baserade och den HGA-baserade på Sudokupussel, och efter varje försök sparas antalet generationer varje algoritm behövde för att hitta lösningen. Den data

som erhålls analyseras med statistiska metoder för att undersöka om skillnader mellan algoritmerna är statistiskt säkerställda eller om de kan förklaras av den statistiska osäkerheten.

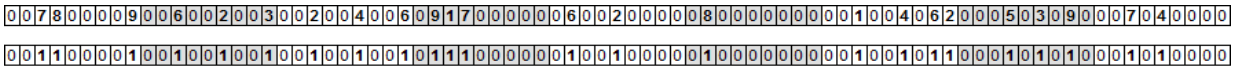
Hypotes två undersöks genom att använda Hammingavståndet för att beräkna skillnaden mellan individer. Hammingavståndet väljs för att arrayerna är lika långa, operationer som appliceras på arrayerna tillåter inte att den första karaktären tas bort och en karaktär läggs till på slutet vilket leder till att Levenshteinavståndet blir missvisande. Hammingavståndet används även av Chen, et al. (2017) och är enligt Morrison och De Jong (2002) ett av de vanligaste måttet för att mäta diversitet på genotyp-nivå. När sökningen har hittat en lösning på Sudokupusslet, beräknas Hammingavståndet mellan de individer som finns i den population som genererades sist. Ett större värde innebär att populationen har en större diversitet än en population med ett mindre Hammingavstånd.

Det sista experimentet undersöker hypotes tre genom att lösa svåra Sudokupussel med den LBSA-baserade och den HGA-baserade algoritmen. Data som sparas är antalet lyckade försök för varje algoritm.

För att undersöka om en grupp med data är större än en annan grupp av data används Wilcoxon *Signed Ranked Test* (Wilcoxon, 1945) tillsammans med Mann-Whitney (Mann och Whitney, 1947). Testerna används för att de inte kräver normalfördelad data och att de går att tillämpa på en datamängd med få värden.

5.1 Applikationsdetaljer

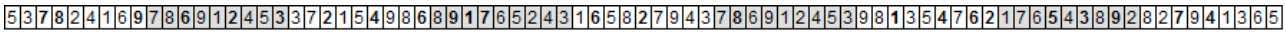
Innan experimenten påbörjades gjordes en förstudie för att försöka hitta de bästa parametrarna för respektive algoritm. Gemensamt för de två algoritmerna är representationen av individer och bygger på en princip som användes av Mantere och Koljonen (2007), varje individ har två arrayer med 81 positioner som motsvarar en cell i en Sudoku matris. Den ena arrayen innehåller siffrorna i Sudoku-pusslet och innehåller de fasta siffrorna och nollor som markerar tomma positioner den andra arrayen innehåller nollor för att visa tomma positioner och ettor som markerar positioner med fasta siffror.

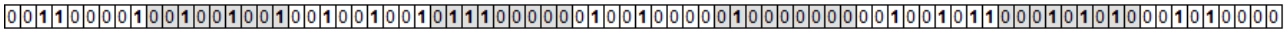


Figur 5.1.1: Representation av Sudoku.

Figur 5.1.1 visar ett exempel på hur arrayerna för en individ som har läst in ett Sudoku ser ut, den övre arrayen innehåller de fasta siffrorna och nollor som senare kommer skrivas över med ett värde mellan ett och nio, den undre innehåller ettor som markerar på vilka positioner de fasta siffrorna finns och nollor för att visa vilka positioner som kan skrivas över.

En viktig egenskap representationen har är att siffrorna i Sudokupusslets boxar läses in från vänster till höger, och sparas efter varandra. De nio första positionerna är den övre vänstra boxen, de nio näst följande positionerna är den övre mittersta boxen etc. När algoritmerna fyller i arrayerna innehåller varje subarray som representerar en box i Sudoku-pusslet alltid siffrorna ett till nio (Figur 5.1.2). En subarray är nio efterföljande positioner och den första subarrayen börjar på position noll.





Figur 5.1.2: Representation av ifyllt Sudoku.

5.2 HGA-baserad jämförelsealgoritm

Jämförelsealgoritmen är baserad på HGA men flera större förändringar på grund av oklarheter kring en del detaljfrågor i Deng och Li (2011).

5.2.1 Selektionsstrategi

Innan algoritmen går in i selektionsfunktionen halveras populationen och bara den första hälften av individerna behålls. Selektionsstrategin väljer ut lika många par av individer som finns i populationen efter

halveringen. Exempelvis om en startpopulation har fyrtio individer halveras den till tjugo, sedan väljs tjugo par ut och varje par genererar en ny individ genom *crossover*-operationen.

```
for i ← 0, i < population size
   identifier ← random integer from 1 to randMax

   if identifier < 6
      upperBound ← 6

   else if identifier < 11
      upperBound ← 11

   else if identifier < 16
      upperBound ← 16

   else
      upperBound ← 20

   endif


   x1 ← random individual with ordinal number less than 'upperBound'
   x2 ← random individual with ordinal number less than 'upperBound'

   while x1 is same individual as x2

       x2 ← random individual with ordinal number less than 'upperBound'

   endWhile

   crossover(x1, x2)

endfor
```

Algoritm 5.2.1: Pseudokod för den HGA-baserade algoritmens selektionsstrategi.

I original HGA beskriver Deng och Li (2011) hur de delar upp den resterande hälften av individerna i fyra kategorier där individerna med lägst fitnessvärde är i den första kategorin, individerna med de näst sämsta fitnessvärden är i den andra kategorin, etc. Selektionen väljer sedan en kategori ur vilken två individer ska väljas, valet av de två individerna utförs sedan slumpmässigt. Variabeln *identifier* används för att välja vilken kategori som ska användas på grund av oklarheter hur *identifier* generas i original HGA, används en modifierad selektionsfunktion som genererar ett värde för *identifier* genom att slumpa ett integer värde från ett till *randMax* som är en konstant variabel som sätts i början av körningen. Under förstudien hittade algoritmen flest lösningar när *randMax* sattes till värden mellan 30 och 40.

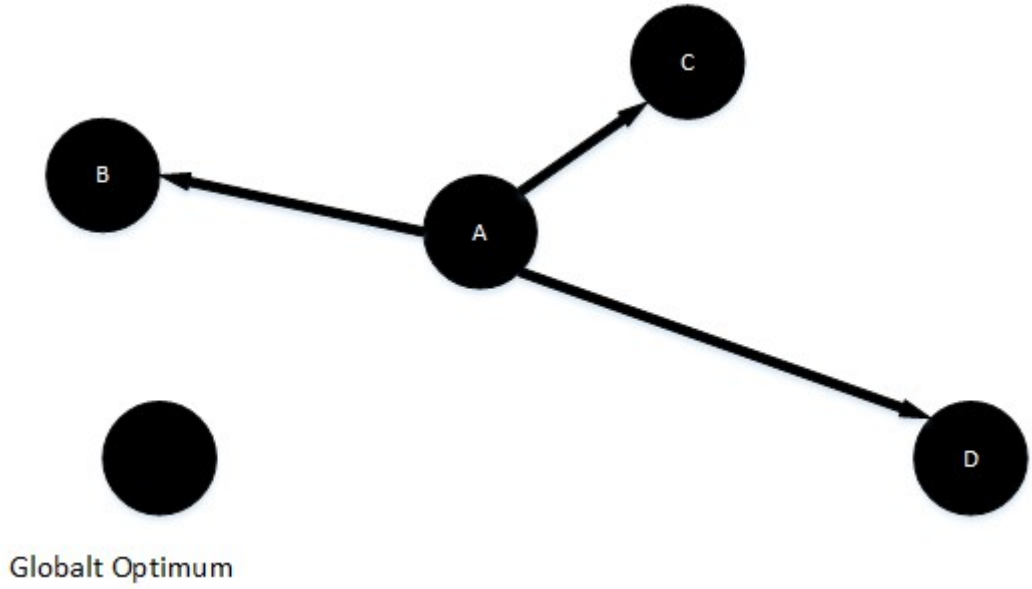
När *identifier* har genererats jämförs värden mot gränsvärden för de fyra kategorierna exempelvis om *identifier* slumpas till åtta, kommer den första och den andra kategorin av individer att användas eftersom värdet är större än fem men mindre än elva. Två individer slumpas sedan fram från kategori ett eller två om samma individ slumpas fram två gånger i samma par väljs en av dem om. Algoritm 5.2.1 visar pseudokod för ersättningsstrategin. Försök gjordes att kontakta Deng och Li (2011) för att få tillgång till koden men något svar erhölls inte.

5.2.2 Crossover-operator

Crossover-operatorn i den HGA-baserade algoritmen fungerar genom att kombinera den bästa och de två slumpvis valda individer från selektionsoperatorn. Operatorn kopierar ett slumpat antal av subarrayerna från den individ som har lägst fitnessvärde av de två utvalda individer till samma subarray i den bästa individen. Om subarray ett och tre väljs skrivs värde i den bästa individen över med värdena från den utvalda individen. Sedan väljs ett slumpat antal subarrayer från den andra utvalda individen men inte samma som valdes från den första och de raderna skrivs in i den bästa individen. För en mer detaljerad beskrivning se 3.2 Crossover operator i Deng och Li (2011).

5.2.3 Mutationsoperator

Crossover-operatorn producerar en ny individ som muteras enligt mutationsoperatorn. En stor skillnad mellan den HGA-baserade och originalversionen är att en förstärkt lokal sökning används i den HGA-baserade algoritmen.



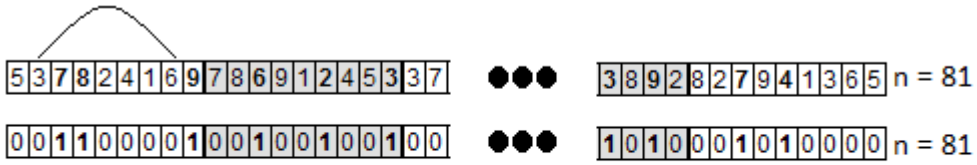
Figur 5.2.1: Den HGA-baserad algoritmen använder lokal sökning.

De svarta cirklarna representerar individer, A är den nya individ som är resultatet av crossover-operatorn, B, C och D är resultatet av mutationsoperatorn. A muteras tre gånger men bara den individ som har lägst fitnessvärde och ligger närmast det globala optimat behålls och läggs till i populationen, i detta fall individ B. Individ A kan inte läggas till i populationen eftersom omuterade individer orsakar en för homogen population.

```
if fitness of best individual > 10
      mutations ← 1

else if fitness of best individual > 5
      mutations ← random integer from 1 to 4

else
      if identifier > 15
          mutations ← random integer from 1 to 2

      else if identifier > 5
          mutations ← random integer from 1 to 2
      else
           mutations ← random integer from 1 to 6
endif
```

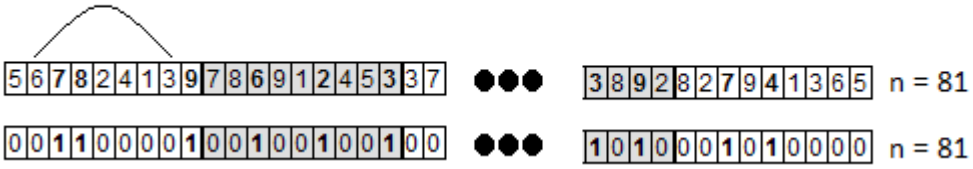
Algoritm 5.2.2: Pseudokod för den HGA-baserade algoritmens mutations-operator.

Antalet mutation som utförs beror på fitnessvärdet hos den bästa individen i den nuvarande populationen (se algoritm 5.2.2), om fitnessvärdet för den bästa individen är högre än tio utförs en mutation för varje ny individ, är värdet mellan fem och tio slumpas antalet mutationer mellan ett och fyra. När sökningen har hittat en individ som har fitnessvärdet fem eller lägre varieras antalet mutationer beroende på vilka kategorier som används för att generera nya individer. Värdet på variabeln *identifier slumpas* fram under selektionen och avgör från vilka kategorier individer kan väljas, och även mellan vilka intervall antalet mutationer ska slumpas.

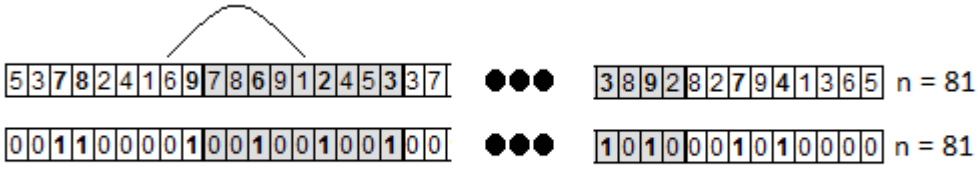


Figur 5.2.2: HGA-baserad algoritm mutation 1. Överst en individs Sudokuarray med storlek n, underst en individs hjälparray med storlek n.

Mutationerna utförs som som i original versionen av HGA. En position som har en nolla i hjälparrayen och är flyttbar väljs slumpvis i arrayen som representerar Sudokupusslet (figur 5.2.2), och byter plats med en annan position inom samma subarray (figur 5.2.3).



Figur 5.2.3: HGA-baserad algoritm mutation 2.Överst en individs Sudokuarray med storlek n, underst en individs hjälparray med storlek n.



Figur 5.2.4: HGA-baserade algoritm felaktig mutation. Överst en individs Sudokuarray med storlek n, underst en individs hjälparray med storlek n.

Figur 5.2.4 demonstrerar ett felaktigt byte eftersom siffror inte kan skiftas mellan två olika subarrayer, de nio första siffrorna tillhör en annan subarray än de nio efterföljande siffrorna. Siffror som har markerats som fixerade siffror kan inte heller byta plats, hjälparrayen markerar fixerade siffror med en etta på samma index som de.

5.2.4 Ersättningsstrategi

I den HGA-baserade algoritmen är de två bästa individerna garanterade att föras vidare till nästa generation. Eftersom den bästa individen ersätts med den näst bästa om den har varit oförändrad under fyra efterkommande generationer, sparas också den näst bästa om inte finns det risk att exempelvis den sämsta individen slumpas till position två och blir den bästa individen när fitnessvärdet försämras.
De övriga individerna i populationen ordnas slumpmässigt, sedan sparas den första halvan av populationen, individerna med ordningsnummer från det första till det mittersta ordningsnumret.

5.3 LBSA-baserad algoritm

Den ursprungliga LBSA algoritmen är designad för att arbeta med flyttalsarrayer, vilket innebär att mutations- och *crossover*-operatorn som används inte kan appliceras på Sudokuproblemet utan operatorerna

måste först anpassas för problemet. Under implementationen måste därför i vissa fall avväganden göras om det är viktigare att göra operatorerna bra på att lösa Sudokupussel eller om det är viktigare att implementera dem nära den ursprungliga LBSA. Valet gjordes att försöka implementera dem likt LBSA för att undersöka om operatorerna som används av LBSA kan lösa Sudokuproblemet.

5.3.1 Initiering av startpopulationerna

Under initieringsfasen skapas den historiska populationen och den första populationen, populationerna fylls med antalet individer som bestäms av parametern för populationsstorlek. När alla individer har skapats fylls deras Sudokuarrayer med slumpade siffror mellan ett och nio tills alla tomma positioner har en siffra. Villkoret att alla subarrayer har siffrorna ett till nio exakt en gång efterföljs under initieringsoperationen.

5.3.2 Selektionsstrategi

LBSA utför inte selektion till *crossover*-operatorn vilket är vanligt i andra evolutionära algoritmer istället skapas en ny försökspopulation och varje individ i den population genomgår *crossover*-processen, men LBSA utför en ersättning av den historiska populationen. I varje iteration kan den historiska populationen med en viss sannolikhet ersättas av den nuvarande populationen.

```
if (a < b | a, b ~ U(0, 1) and iteration > 20)

      oldP ← P

endif
```

Algoritm 5.3.1: Ersättning av historisk population LBSA-baserad algoritm.

Algoritm 5.3.1 visar selektionsstrategin för ersättning av den historiska populationen *oldP* med den nuvarande populationen *P*. Den LBSA-baserade algoritmen inför villkoret att sökningen måste vara inne på minst den tjugoförsta iterationen innan ersättning av den historiska populationen kan ske. Detta på grund av att under initieringen skapas två olika populationer, om den historiska populationen ersätts under den första iterationen blir de identiska. Om populationerna inte är identiska ökar det chansen att individerna blir mer olika varandra när information från den historiska populationen används under *crossover*-operationen. Efter ett fåtal iterationer kommer individernas fitnessvärde i den nuvarande populationen vara bättre än individernas fitnessvärde i den historiska populationen. Det är då bättre att ersätta den historiska populationen än att tillåta sämre individer föra information vidare.

5.3.3 Mutationsoperator

Den implementerade mutations-operatorn är ett försök att anpassa mutationsekvationen från den ursprungliga LBSA algoritmen till Sudokuproblemet och den representation av individer som används av den LBSA-baserad algoritmen. LBSA modifierar mutations-operatorn från BSA som används för att generera försökspopulationen, den LBSA-baserade algoritmen använder på samma sätt mutationen för att generera sin försökspopulation. Mutations-operatorn i LBSA kopierar information från en kombination av antingen den nuvarande populationen, den historiska population och den bästa individen (ekvation 2.3.2), eller populationen, en slumpad individ, och den sämsta individen (ekvation 2.3.3) till en ny mutantindivid. Vilken av ekvationerna som används för en individ avgörs genom slump. Det är inte önskvärt att addera eller subtrahera värden i Sudokupussel eftersom det kommer att skapa värden som ligger utanför det tillåtna intervallet. Det som används från LBSA i den LBSA-baserade algoritmen är därför de olika entiteterna som används i mutationen, i den första mutationsvarianten kopierar algoritmen subarrayer från populationen, den historiska populationen och den bästa individen till försökspopulationen och i den andra varianten kopieras subarrayer från populationen, en slumpad individ i populationen, den bästa individen och den sämsta individen till försökspopulationen. Antalet subarrayer som kopieras från varje entitet i mutationen bestäms slumpmässigt.

```
for i < size of trial population

    if(a < b|a, b ~ U(0, 1))

       for x ← rand(0, 9)
              index ← random index
              T[i].setRegion(index, P[i].getRegion(index))

       for y ← boundryControl((rand(0, 9 - x))
              index ← random index
              T[i].setRegion(index, oldP[i].getRegion(index))

       for z ← boundryControl(rand(0, 9 - y))
              index ← random index
              T[i].setRegion(index, best.getRegion(index))
    else

       for x ← rand(0, 9)
              T[i] ← random index
              T[i].setRegion(index, P[k].getRegion(index))

       for y ← boundryControl(rand(0, 9 - x))
              index ← random index
              T[i].setRegion(index, best.getRegion(index))

       for z ← boundryControl(rand(0, 9 - y))
              index ← random index
              T[i].setRegion(index, P[i].getRegion(index))

       for v ← boundryControl(rand(0, 9 - z))
              index ← random index
              T[i].setRegion(index, worst.getRegion(index))
   endif

endfor
```

Algoritm 5.3.2: Mutations-operatorn i den LBSA-baserade algoritmen.

Algoritm 5.3.2 innehåller pseudokod för mutations-operatorn som används av den LBSA-baserade algoritmen. Variabeln *T* är försökspopulationen, *P* den nuvarande populationen, *oldP* den historiska populationen, *best* den individ som har bäst fitness värde i den nuvarande populationen, *worst* den individ som har det sämsta fitnessvärdet i den nuvarande populationen, *setRegion* sätter en subarray för en individ till en given subarray, *getRegion* hämtar en subarray från en individ och *boundreControl* kontrollerar att värdet för en variabel inte underskrider det minsta möjliga och inte överskrider det högsta möjliga värdet.

Under mutationen randomiseras vilken av de två mutationsvarianterna som ska användas för varje individ genom påståendet `if(a < b|a, b ~ U(0, 1))`, där *U* står för den uniforma distributionen. För att avgöra hur många subarrayer som kopieras slumpas ett värde mellan noll och nio, sedan kopieras det antalet subarrayer från den första entiteten. Variabeln *index* avgör vilken subarray som ska kopieras. När en array har valts kan den inte väljas igen, exempelvis kan inte subarray med index ett kopieras från individen i populationen till individen i försökspopulationen för att sedan skrivas över med subarray med index ett från den bästa individen.

5.3.4 Crossover-operator

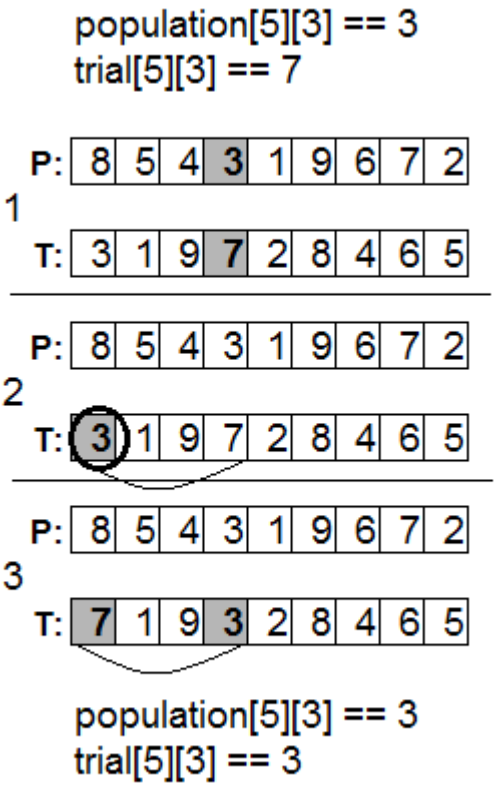
Den LBSA-baserade algoritmen använder samma *crossover*-operator som BSA men den måste anpassas för den representation som används av den LBSA-baserade algoritmen eftersom den inte tillåter att en siffra finns mer än en gång i varje subarray.

```
crossoverMap[trial.size][problem dimension] ← fill with ones

if(a < b|a, b ~ U(0, 1))

      for i < size of trial population

            for 1 to mixrate * rand * problem dimension

                  crossoverMap[i][random(problem dimension)] ← 0

            endfor

      endfor

else

      crossoverMap[i][random(problem dimension)] ← 0

endif

for i < size of trial population

      for j < problem dimension

            if crossoverMap[i][j] == 1 and j is not fixed

                  T[i][j] ← P[i][j]

            endif

      endfor

endfor
```

Algoritm 5.3.3: Crossover-operatorn i den LBSA-baserade algoritmen.

Algoritm 5.3.3 innehåller pseudokod för *crossover*-operatorn för den LBSA-baserade algoritmen. Variabeln *T* är försökspopulationen, *P* den nuvarande population och *fixed* är en position som inte får byta plats i Sudokuarrayen.

Först fylls en matris (*crossoverMap*) med måtten antal individer i försökspopulationen x antal tomma rutor i Sudokupusslet, med siffran ett på alla positioner. Sedan väljs en av två varianter för att bestämma antalet positioner som ska förändras under *crossover*-operatorn. Den första varianten sätter ett antal celler i matrisen med nollor, det maximala antalet bestäms av *mixrate* som är en konstant som bestäms i början av sökningen. Den andra varianten sätter ett slumpat index i matrisen till noll. När en av de två varianterna har utförts går funktionen vidare till *crossover*-operationen. Den itererar över alla positioner i matrisen och för varje position som innehåller en etta, kopieras värdet som finns på den motsvarande positionen (*j*) för den aktuella individen (*i*) från populationen till försökspopulationen. Eftersom varje subarray i representationen av Sudokupusslen måste innehålla en siffra från ett till nio exakt en gång, kan inte *crossover*-operatorn skriva över en bit i försökspopulationen med en bit i den nuvarande populationen. Värdet som finns på den aktuella positionen i försökspopulationen måste flyttas till en annan position inom samma subarray.



Figur 5.3.1: Swap-operationen som utförs av crossover-operatorn i den LBSA-baserad algoritmen.

Figur 5.3.1 illustrerar den utförda operationen, *P*-arrayen representerar en subarray från en individ i populationen och *T*-arrayen en subarray från en individ i försökspopulationen. I steg ett bestäms en slumpad position i subarrayen i figuren väljs position tre i individerna på position fem i populationen och försökspopulationen. Värdet som finns på den positionen i populationen ska efter crossover-operationen finnas på samma position i försökspopulationen men om det skrivs in direkt finns siffran tre två gånger inom samma subarray vilket representationen inte tillåter. I steg två identifieras därför den position som innehåller det värde som finns på den aktuella positionen i populationen och i steg tre utförs en swap-operation för att placera siffran tre på rätt plats samtidigt som det ursprungliga värdet i försökspopulationen behålls i subarrayen.

5.3.5 Fylloperator

Efter mutation och *crossover* kan det finnas individer i försökspopulationen som har tomma positioner i deras Sudokuarray. Tomma positioner uppstår om det totala antalet kopierade subarrayer från andra individer till försöksindividen understiger nio och försöksindividens tomma subarrayer inte har fylls av *crossover*-operatorn.

```
for i < size of trial population

      for j < size of sudokuArray

            if sudokuArray[j] is empty

                  sudokuArray[j] ← generate random(1-9) not in
                                   current subarray

            endif

      endfor

endfor
```

Algoritm 5.3.4: Funktion för att fylla tomma positioner efter mutation och crossover.

Fylloperatorn (algoritm 5.3.4) itererar över varje individ i försöks populationen och över varje bit i individernas Sudokuarray. När en tom ruta erhålls genereras en randomiserad siffra som inte finns i den aktuella subarrayen som indikeras av index *j* och siffran skrivs in i index *j*. Exempelvis om index *j* är från noll till åtta arbetar funktionen med den första subarrayen, om *j* är från nio till sjutton arbetar funktionen med den andra subarrayen osv. Fylloperatorn är den funktionen som i implementationen har den högsta komplexiteten.

5.3.6 Ersättningsstrategi

Den LBSA-baserade algoritmen använder en elitistisk ersättningsstrategi där individer i försökspopulationen jämförs med individer på motsvarande position i den nuvarande populationen, om individen i populationen har ett högre värde än försökspopulationen ersätts den med försöksindividen.

```
for i < T.size

    if fitness(T[i]) <= fitness(P[i]) && P[i] != best

        P[i] ← T[i]

    else if fitness(T[i]) < fitness(P[i])

        P[i] ← T[i]

    endif

endfor
```

Algoritm 5.3.5: Ersättningsstrategi för den LBSA-baserade algoritmen.

Algoritm 5.3.5 innehåller pseudokoden för ersättningsstrategin, *T* är försökspopulationen, *P* är den nuvarande populationen, *best* är den individ med det bästa fitnessvärdet i den nuvarande populationen och *fitness* returnerar fitnessvärdet för en individ. Individer i försökspopulationen jämförs med individer i populationen ersättning sker när villkoret lägre eller lika fitness är uppfyllt. Med undantaget att om individen är den bästa i populationen måste försöksindividen fitnessvärde vara lägre. Anledningen till att det måste vara lägre är att den fitnessvärdet för den bästa individen försämras om den har varit den bästa efter ett bestämt antal iterationer. Om den skrivs över av en individ som är identisk nollställs antalet genomförda iterationer men individen är oförändrad.

6 Resultat och analys

Innan experimenten utfördes genomfördes en förstudie per algoritm för att identifiera under vilka parametrar som algoritmerna hade den högsta lösningsgraden. På grund av att experimenten krävde mycket mer tid än planerat och att algoritmerna presterade med en lägre lösningsgrad än förväntat utfördes endast de experiment som använde de enkla Sudoku-pusslen. Under en förstudie med medelsvåra och svåra pussel

klarade ingen av algoritmerna att hitta några lösningar, den ursprungliga versionen av HGA har en lägre lösningsgrad för svåra pussel än för lätta. En lägre lösningsgrad för svåra pussel var därför förväntad men att algoritmen inte klarade att lösa några svårare pussel var inte det förväntade resultatet. Troliga orsaker till att den HGA-baserade algoritmen inte klarade att lösa svåra Sudokupussel, är förändringarna som gjorts på selektionsoperatorn och att den lokala sökningen gör att populationen snabbare sammanstrålar. Den snabbare sammanstrålningen gör att populationen inte bibehåller diversiteten. För LBSA behövs troligen implementationen optimeras för Sudokuproblemet för att algoritmen ska kunna lösa svåra Sudokupussel.

6.1 Lösningsgrad

Det första delmålet var att fastställa hur LBSA och HGA skiljer sig med avseende på antalet generationer som krävs för att lösa Sudokupussel, och undersöktes genom att applicera implementerade varianter av algoritmerna på fem olika Sudokupussel. Undersökningen delades upp i fem experiment där det första experimentet med experimentnummer ett applicerar båda algoritmerna på pussel ett trettio gånger, det andra experimentet med experimentnummer två applicerar båda algoritmerna på pussel två trettio gånger, etc. Två tabeller presenterar hur de båda algoritmerna presterade med avseende på lösningsgrad och antal generationer som krävdes för att hitta en lösning, en graf används för att illustrerar skillnaden mellan algoritmerna.
Tabell 6.1.1 visar en sammanställning av experimenten som utfördes med den HGA-baserade algoritmen. Kolumn ett innehåller experimentets id-nummer, kolumn två visar antalet försök, kolumn tre visar procentsatsen av de lyckade försöken, kolumn fyra visar antalet lyckade försök, kolumn fem visar medelvärdet för *n*, kolumn sju visar det minsta värdet för *n* och kolumn sju det största värdet för *n*. Där *n* är antalet generationer som krävdes för att hitta lösningen eller om ingen lösning hittas antalet utförda iterationen innan sökningen avbryts.

Experiment ID	Antal försök	Andel lyckade försök	Antal lyckade försök	avg(n)	min(n)	max(n)
1	30	87%	26	2867	75	9595
2	30	90%	27	1613	93	9706
3	30	17%	5	3429	1267	5210
4	30	23%	7	2970	329	8479
5	30	30%	9	2314	74	4271

Tabell 6.1.1: Sammanställning av resultaten för den HGA-baserade algoritmen.

Tabell 6.1.2 visar en sammanställning av experimenten som utfördes med den LBSA-baserade algoritmen. Kolumn ett innehåller experimentets id-nummer, kolumn två visar antalet försök, kolumn tre visar procentsatsen av de lyckade försöken, kolumn fyra visar antalet lyckade försök, kolumn fem visar medelvärdet för *n*, kolumn sju visar det minsta värdet för *n* och kolumn sju det största värdet för *n*. Där *n* är antalet generationer som krävdes för att hitta lösningen eller om ingen lösning hittas antalet utförda iterationen innan sökningen avbryts.

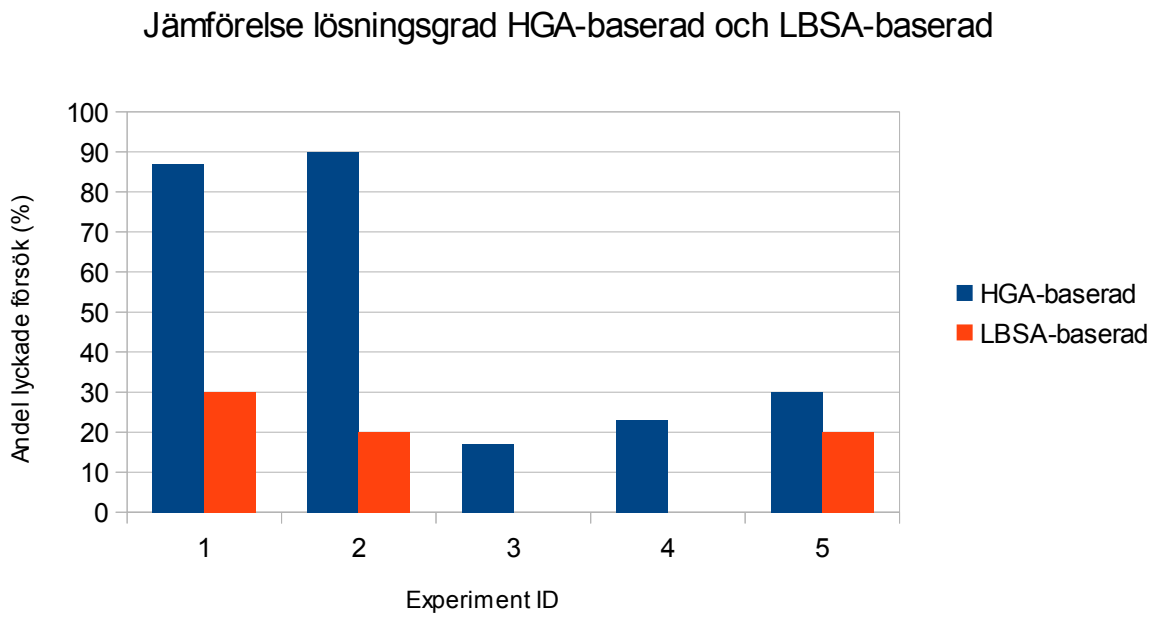
Experiment ID	Antal försök	Andel lyckade försök	Antal lyckade försök	avg(n)	min(n)	max(n)
1	30	30%	9	1959	290	6596
2	30	20%	6	2434	915	4362
3	30	0%	-	-	-	-
4	30	0%	-	-	-	-
5	30	20%	6	4184	690	7275

Tabell 6.1.2: Sammanställning av resultaten för den LBSA-baserade algoritmen.

Eftersom skillnaderna mellan algoritmerna är stor och speciellt den LBSA-baserade algoritmen har en låg lösningsgrad är det svårt att göra jämförelsen mellan antalet generationer som algoritmerna kräver för hitta en lösning. Den HGA-baserade algoritmen har en betydligt bättre lösningsgrad, hittar i de bästa fallen lösningen på ett mindre antal generationer och klara att hitta några lösningar under alla experiment.

Den HGA-baserade algoritmen lyckas lösa alla fem Sudoku-pusslen bättre än den LBSA-baserade, pussel ett

löses 57 procent mer frekvent, Sudoku två 70 procent mer frekvent. Den LBSA-baserade algoritmen klarar inte att lösa pussel tre och fyra under det bestämda antalet generationer. Pussel fem klarar Den HGA-baserade och den LBSA-baserade algoritmen med en liknande lösningsgrad.



Figur 6.1.1: Stapeldiagram över lösningsgraden för den HGA-baserade och den LBSA-baserade algoritmen.

Figur 6.1.1 ger en bild av hur algoritmerna presterar jämfört med varandra. Den HGA-baserade algoritmen klara att lösa speciellt de två första Sudokupusslen mycket mer frekvent än den LBSA-baserade. Båda algoritmerna har en sämre lösningsgrad för de tre sista Sudokupusslen än för de två första. Under det tredje och fjärde experimentet klara inte den LBSA-baserade algoritmen att lösa några Sudokupussel. Den låga lösningsgraden för den LBSA-baserade algoritmen tyder på att den inte är tillräckligt anpassad för Sudokuproblemet, andra hybrida evolutionära algoritmer har en hundraprocentig lösningsgrad för lätta Sudokupussel exempelvis Wang et. al. (2015), och Mantere och Koljonen (2007). Om den LBSA-baserade algoritmen anpassas för Sudokuproblemet är det möjligt att lösningsgraden ökar, hur mycket av algoritmen som behöver förändras behöver testas med fler experiment.

Andelen lösta Sudokupussel kan ökas genom att höja det maximalt tillåtna antalet iterationer men på grund av tiden som krävs för att köra det redan bestämda maximala antalet blir det problematiskt.

6.2 Diversitet

Ekvation 6.2.1 visar den valda definitionen av diversitet hos en population, *d* står för diversitet och *hamming* är Hammingavståndet för en population.

$$d = avg\left(hamming\left(population\right)\right) \qquad \textit{Ekvation 6.2.1}$$

För att uppskatta diversiteten i en population uppmättes skillnaden mellan alla individer i en slutpopulation med Hammingavståndet, sedan beräknas medelvärdet för individerna. Slutligen utfördes Wilcoxon *Signed Ranked Test* (Wilcoxon, 1945) och Mann-Whitneys U-test (Mann och Whitney, 1947) för att undersöka om skillnaden är statistiskt säkerställd.

Tabell 6.2.1 innehåller resultaten av jämförelsen över vilken algoritm som bäst klarade av att bibehålla diversiteten i populationen. Där *d* är diversiteten. Kolumn ett innehåller experimentets identifikationsnummer, kolumn två och fyra det minsta värdet för diversiteten bland alla de sista populationerna för experimenten. Den tredje och sjätte det största värdet för diversiteten bland alla de sista populationerna för experimenten. Den fjärde och sjunde kolumnen medelvärdet för diversiteten bland de

sista populationerna för experimenten. Den åttonde kolumnen innehåller P-värdet och den nionde kolumnen innehåller Z-värdet. För det första experimentet går det inte att säkerställa att den HGA-baserade algoritmen har en högre diversitet än den LBSA-baserade algoritmen i slutpopulationen eftersom Z-värdet är lägre än 1.64. Det går under de fyra resterande experimenten statistiskt säkerställa att den LBSA-baserade algoritmen har en högre diversitet än den HGA-baserade.

	HGA-baserad algoritm			LBSA-baserad algoritm				
Exp. ID	min(d)	max(d)	avg(d)	min(d)	max(d)	avg(d)	P-värde	Z-värde
1	7,63	21,45	**13,03**	3,13	22,87	11,62	0,91	1,33
2	8,19	16,27	12,33	6,76	29,30	**16,32**	1,00	-3,35
3	2,93	20,72	10,69	8,56	22,21	**16,39**	1,00	-4,48
4	6,72	21,87	10,59	14,21	26,71	**20,69**	1,00	-5,94
5	7,59	18,51	10,93	5,35	28,02	**17,38**	1,00	-5,62

Tabell 6.2.1: Medelvärdet för antal positioner som en individ skiljer sig från andra individer.

Den LBSA-baserade algoritmen har lägre diversitet i det första experimentet jämfört med de övriga experimenten vilket troligen beror på parameterinställningar. Under det första experimentet lärde sig individer mer från den individ med bäst fitnessvärde än under de andra experimenten. De parameterinställningarna fungerade inte för de andra experimenten, lösningsgraden var låg eller nollprocentig och populationerna fick låg diversitet ibland med helt identiska individer. Det ledde till att förändringar av parametrarna gjordes för förhindra att populationerna förlorade sin diversitet och för att försöka förbättra lösningsgraden. Under de resterande fyra experimenten har den LBSA-baserade algoritmen en jämnare diversitet.

Den LBSA-baserade algoritmen har även en något lägre diversitet i det första experimentet jämfört med den HGA-baserade algoritmen, men det kan bero på fördelningen av datavärdena eftersom Z-värdet är lägre än 1.64. För de övriga experimenten är diversiteten högre för den LBSA-baserade algoritmen än den HGA-baserade. Att den HGA-baserade algoritmen har lägre diversitet beror på att individer får mer information från den bästa individen och att den lokala sökningen gör att algoritmen snabbare sammanstrålar mot ett optimum. Under förstudien hade den HGA-baserade algoritmen en lägre lösningsgrad utan lokalsökning, för vissa Sudokupussel var lösningsgraden noll procent.

6.3 Svåra Sudokupussel

Det tredje delmålet var att undersöka om LBSA kunde lösa fler svåra pussel än HGA, men under förstudien klarade ingen av algoritmerna att lösa några svåra pussel. Den ursprungliga HGA algoritmen hade problem med svåra Sudokupussel med en lösningsgrad på 17 procent, förändringarna av selektionsstrategin i den HGA-baserade algoritmen påverkade lösningsgraden negativt. Den LBSA-baserade algoritmen måste anpassas till problemdomänen för att lösa svåra Sudokupussel, vilka operatorer som fungerar bra eller dåligt behöver testas med ytterligare experiment. Chel et. al. (2016) har problem med att lösa svåra Sudokupussel deras algoritm har en lösningsgrad på tolv procent, vilket indikerar att det finns utmaningar med att anpassa en evolutionär algoritm till Sudokuproblemet. Wang et. al. (2015) lyckas med en hundraprocentig lösningsgrad lösa svåra Sudokupussel, deras algoritm använder filtrerade mutationer, vilket innebär att siffror under mutationsfasen inte kan flyttas till positioner som är otillåtna för den siffran. Exempelvis kan inte en siffra placeras i samma rad som en annan siffra med samma värde.

7 Diskussion

Under experimentens genomförande visade sig att båda algoritmerna hade det svårare än förväntat att lösa lätta Sudokupussel. Den HGA-baserade algoritmen som under förstudien löste alla Sudokupussel uppnår inte samma lösningsgrad under experimenten. En orsak till att den LBSA-baserade algoritmen har en låg lösningsgrad är skillnaderna mellan egenskaperna för den ursprungliga algoritmens problemdomän och egenskaperna för Sudokuproblemet. Den ursprungliga LBSA algoritmen är designad för att arbeta med problem som använder arrayer med reella tal och använder enkla aritmetiska operationer för att modifiera individer. Efter att individer har generats och modifierats kontrollerar algoritmen att värdena i arrayen som representerar individen ligger inom de tillåtna gränsvärdena och om inte justerar dem genom att

slumpmässigt höja eller sänka dem. Sudokuproblemet har mer komplexa begränsningar som att siffror bara får förekomma en gång per rad, kolumn och box, och Sudokupusslets startsiffror får aldrig modifieras. Den LBSA-baserade algoritmen har en större diversitet i slutpopulationen än den HGA-baserade i fyra av fem experiment. Även under experiment två där den HGA-baserade algoritmen har en lösningsgrad på 90%, vilket innebär att den LBSA-baserade algoritmen utför fler iterationer än den HGA-baserade, har den LBSA-baserade algoritmen en högre diversitet. Det indikerar att algoritmen har en för långsam sammanstrålningshastighet för att hitta lösningen på Sudokupussel inom det maximala antalet iterationer. Ytterligare experiment behöver utföras för att undersöka vilka operatorer som behöver förändras eller bytas ut, för att den LBSA-baserade algoritmen ska få en högre lösningsgrad. En intressant undersökning är att se hur diversiteten utvecklas under en sökning genom att beräkna Hammingavståndet en population efter exempelvis var hundrade generation, för att sedan undersöka den hastighet med vilken diversiteten minskar. Ytterligare en intressant jämförelse som kunde gjorts är att jämföra hur diversiteten skiljer sig mellan algoritmerna för samma generation. Troligen kommer den HGA-baserade algoritmen ha mindre diversitet bland individerna som har bra fitnessvärde än den LBSA-baserade, eftersom den HGA-baserade applicerar mer elitism. Det är troligt att diversitet för hela populationen minskar snabbare per generation för den HGA-baserade, medans den LBSA-baserade håller en jämnare diversitet och diversiteten minskar mindre per generation. Författaren anser att det är intressant att undersöka algoritmer som försöker bibehålla diversiteten i populationen, eftersom exempelvis Segura et. al. (2016) har implementerat en algoritm som klarar att lösa svåra Sudokupussel utan att algoritmen är anpassad till problemdomänen. Den designade algoritmens målsättningsfunktion tar hänsyn både till hur individer påverkar fitnessvärdet och populationens diversitet, vilket leder till att populationen inte konvergerar för tidigt. Algoritmen har en lägre sammanstrålningshastighet än andra algoritmer men är den första icke hybrida evolutionära algoritm som framgångsrikt klarar att lösa det Sudokupusslet som anses vara det svåraste.

Resultaten för den LBSA-baserade algoritmen kan inte generaliseras till den ursprungliga algoritmen, eftersom förändringarna gör att den LBSA-baserade algoritmen inte klarar av att lösa de problem den ursprungliga algoritmen klarar. Detta eftersom den LBSA-baserade algoritmen inte minimerar tal utan byter plats på och skriver siffror i en arrays tomma positioner. Den ursprungliga algoritmen behöver på något sätt anpassas för problemdomänen för att var användbar inom området, om för stora förändringar måste göras kan det betyda att algoritmen egentligen inte går att anpassa till problemdomänen.

I projektets början var tanken att implementera en LBSA-baserad algoritm som kunde säga något om LBSA kunde anpassas för Sudokuproblemet. Under designfasen av algoritmen visade det sig att stora förändringar av flera operatorer var nödvändiga, för att den LBSA-baserade algoritmen skulle klara av att lösa Sudokupussel. Designvalet gjordes att försöka göra minimala förändringar av den ursprungliga algoritmen, för att undersöka om operatorerna som de var designade var användbara för Sudokuproblemet. De operatorerna som kunde användas från den ursprungliga LBSA algoritmen var selektion-I, *crossover*-operatorn och selektion-II, initiering av populationerna, mutationsoperatorn och kontrollen av gränsvärden var tvungna att anpassas till att istället för att utföra aritmetiska operationer utföra byten mellan positioner i Sudokuarrayerna. Det hade varit bättre att optimera operatorerna för Sudokuproblemet eller att använda en algoritm som visat sig ha en hög lösningsgrad för Sudokuproblemet och sen använda en operator från den ursprungliga LBSA algoritmen, för att undersöka hur den operatorn påverkar lösningsgraden. Ett annat alternativ hade varit att designa en algoritm som använder mutationsoperatorn från LBSA, men en *crossover*-operator lik den som används av HGA. Subarrayer hade kopierats från de olika individerna till den nya mutanten. Sedan hade algoritmen applicerat en konventionell mutationsoperator som byter plats på ett slumpat antal positioner. Med en låg sannolikhet att en individ muteras. Det hade varit möjligt att inkludera mer avancerade tekniker som filtrerade mutationer, som inte tillåter att siffror flyttas till positioner som algoritmen vet är felaktiga.

En faktor som hade oväntad betydelse var skillnaden mellan Sudokupussel även inom samma svårighetsgrad. De pussel som användes under experimenten laddades ner från samma hemsida (https://www.sudoku-solutions.com/, 2017) och var alla från kategorin *simple*. En analys av Sudokupusslen visar att de givna startsiffrorna för Sudokupussel ett till fem var 33, 30, 30, 31 och 31, de som var lättast att lösa var pussel ett med 33 givna siffror och pussel två med 30 givna siffror. Placeringen av siffror i rader, kolumner och underboxar var liknande vid jämförelse av de olika pusslen. Även om skillnader i placeringen fanns var skillnaden mellan pussel ett och två större än exempelvis mellan pussel två och fyra. Det gick inte att identifiera någon egenskap hos pusslen som kan förklara varför lösningsgraden varierade kraftigt mellan

experimenten. Fem lätta Sudokupussel är inte tillräckligt för att representera alla möjliga lätta Sudokupussel, och experimentet kunde ha utökats med ytterligare pussel. Skillnaden på lösningsgraden tyder på att Sudokupussels utformning har en påverkan av resultatet, även om de har ett liknande antal givna positioner och tillhör samma svårighetsgrad. Fler Sudokupussel tillåter experimentet att utforska en större del av de existerande Sudokupusslen, men på grund av tiden den LBSA-baserade algoritmen kräver för att utföra ett försök ställdes antal Sudokupussel mot antal försök. Exempelvis om antalet Sudokupussel ökas till tio kan algoritmerna bara tillämpas femton gånger per pussel. Den stora skillnaden mellan Sudokupussel med liknande antal startsiffror och från samma svårighetsgrad var oförväntad, när experimenten var utförda fanns det inte tid att testa på fler pussel. Om detta var känt från början hade fler pussel valts och försöken minskats något, ett alternativ hade varit sju pussel och tjugo försök. Utförs färre än tjugo försök anser författaren risken att resultaten påverkas av slumpen är för stor, det gäller inte bara lösningsgraden utan även jämförelsen av diversiteten.

På grund av den höga komplexiteten för den LBSA-baserade algoritmen är det inte lämplig att applicera algoritmen på Sudokupussel som är större än 9 x 9, mindre pussel borde vara lättare att lösa för den LBSA-baserade algoritmen och inom en kortare tid. Troligen är andra algoritmer som exempelvis den HGA-baserade både bättre på att lösa dem och löser dem inom kortare tid.

8 Validitetshot

Metoden implementation används vilket innebär att ett validitetshot är att LBSA och HGA implementeras felaktigt. Att göra en identisk version av LBSA kan vara svårt eftersom källkoden inte offentliggjorts, men eftersom de viktiga delarna beskrivs i Chen et al. (2017) går det implementera en version av LBSA. Under utvecklingen av den versionen efterföljdes god mjukvaruutvecklingssed för att minimera validitetshotet. Viktiga delar av källkoden tillgängliggörs för att den implementerade versionen av LBSA ska gå att jämföra med den beskrivning som finns i den publicerade artikeln (Chen et al., 2017).

Ett annat hot är att jämförelse mellan saker som inte är jämförbara. För att minska det hotet mot validiteten används identiska fitnessfunktioner för den LBSA-baserade och den HGA-baserade. För att minska validitetshotet att resultat beror på statistisk osäkerhet används statistiska metoder och experiment utförs flera gånger.

9 Slutsats

Den LBSA-baserade algoritmen har i tre av fem experimenten en låg lösningsgrad och i två av fem löser inte algoritmen några pussel. Detta tyder på att den LBSA-baserade algoritmen inte är lämplig att använda för Sudokuproblemet, operatorerna måste anpassas mer till Sudokuproblemet och komplexiteten måste förbättras. Algoritmen utför många operationer som har en hög komplexitet vilket gör den långsammare än den HGA-baserade, den höga komplexiteten beror till stor del på två orsaker. Den första orsaken är att *crossover*-operatorn istället för att skriva över värden i en subarray, utför platsbyten inom arrayen för att garantera att representationens villkor att inga duplicerade värden finns inom samma subarray. Den andra orsaken är att om mutations-operatorn kopierar ett fåtal subarrayer från alla involverade individer till försöksindividen, samtidigt som *crossover*-operatorn endast kopierar en bit till försöksindividen. Finns det många tomma positioner som måste fyllas av fyllnadsoperatorn och den operatorn har den högsta komplexiteten av alla operatorer som används av den LBSA-baserade algoritmen. Om komplexiteten för fyllnadsoperatorn kan förbättras, minskar tiden per iteration för den LBSA-baserade algoritmen markant.

Den låga lösningsgraden för den LBSA-baserade algoritmen indikerar att antingen är LBSA inte lämplig för att lösa Sudokuproblemet, eller att LBSA måste anpassas mer till problemdomänen för att algoritmen ska få en högre lösningsgrad. Om flera stora förändringar måste göras för att den ursprungliga LBSA algoritmen ska vara användbar för Sudokuproblemet, kan det vara bättre att använda en annan algoritm som från början är anpassad för problemet.

Framtida arbete är att kombinera en eller flera operatorer med andra algoritmer för att öka lösningsgraden för Sudokuproblemet, eller som Chen et. al. (2017) skriver undersöka applikationsområden för LBSA, och att ytterligare förbättra LBSAs förmåga att bibehålla diversitet. För att förbättra och undersöka LBSAs förmåga att bibehålla diversiteten, kan det vara intressant att applicera andra diversitets mått för att se hur det

påverkar resultaten.

Referenser

Adcock, A. D., Demaine, E. L., Demaine, M. P., O'Brien, M., Reidl, F., Villaamil, F. D. & Sullivan, B. (2015). *Zig-Zag Numberlink is NP-Complete.* Journal of Information Processing, 23(3), s.239-245.

Asif, M. & Baig, R. (2009). *Solving NP-complete problem using ACO algorithm*. 2009 International Conference on Emerging Technologies. Doi: 10.1109/ICET.2009.5353209.

Berndtsson, M., Hansson J., Olsson, B. & Lundell, B. (2008). *Thesis Projects, A Guide for Students in Computer Science and Information Systems* (2nd. ed) Springer-Verlag London Limited.

Brabazon, A., O'Neill, M. and McGarraghy, S. (2015). *Natural Computing Algorithms*. 1st ed.

Chel, H., Mylavarapu, D. & Sharma, D. (2016). *A novel multistage genetic algorithm approach for solving Sudoku puzzle*. 2016 International Conference on Electrical, Electronics, and Optimization Techniques (ICEEOT). Doi: 10.1109/ICEEOT.2016.7754798.

Chen, D., Zou, F., Lu, R. & Wang, P. (2017). *Learning backtracking search optimisation algorithm and its application* Inf. Sci., 376 (2017), s. 71-94.

Civicioglu, P. (2013). *Backtracking search optimization algorithm for numerical optimization problems.* Appl. Math. Comput., 219 (15) , s. 8121–8144.

Cotta, C. & Troya, J. (1998). *On decision-making in strong hybrid evolutionary algorithms. Lecture Notes in Computer Science*, s.418-427. Doi: 10.1007/3-540-64582-9_772.

Črepinšek, M., Liu, S., & Mernik, M. (2013). *Exploration and exploitation in evolutionary algorithms*. ACM Computing Surveys, 45(3), 1-33.

Demaine, E., Ma, F., Schvartzman, A., Waingarten, E. & Aaronson, S. (2016). *The Fewest Clues Problem,* 8th International Conference on Fun with Algorithms 2016, Leibniz International Proceedings in Informatics (LIPIcs) doi: 10.4230/LIPIcs.FUN.2016.12

Deng, X. & Li, Y. (2011). *A novel hybrid genetic algorithm for solving Sudoku puzzles.* Optimization Letters, 7(2), s.241-257.

Falcone, J. L. (2013). *Scheduling 101: A method to the application of Sudoku game theory to surgical rotation scheduling.* American Surgeon, Volume 79, Issue 9, September 2013, s. 958-960

Jilg, J. (2009). *Sudoku evolution.* 2009 International IEEE Consumer Electronics Society's Games Innovations Conference. Doi: 10.1109/ICEGIC.2009.5293614

Kamal, S., Chawla, S. & Goel, N. (2015). *Detection of Sudoku puzzle using image processing and solving by Backtracking, Simulated Annealing and Genetic Algorithms: A comparative analysis*. 2015 Third International Conference on Image Information Processing (ICIIP). Doi: 10.1109/ICIIP.2015.7414762

Mantere, T. & Koljonen, J. (2009). *Ant colony optimization and a hybrid genetic algorithm for sudoku solvin*g. 15th International Conference on Soft Computing: Evolutionary Computation, Genetic Programming, Fuzzy Logic, Rough Sets, Neural Networks, Fractals, Bayesian Methods, MENDEL 2009, Brno, Czech Republic

Mantere, T., & Koljonen, J. (2007). *Solving, rating and generating sudoku puzzles with GA*. Paper presented at the *2007 IEEE Congress on Evolutionary Computation, CEC 2007,* 1382-1389. doi:10.1109/CEC.2007.4424632

Mann, H. & Whitney, D. (1947). *On a Test of Whether one of Two Random Variables is Stochastically Larger than the Other. The Annals of Mathematical Statistics,* 18(1), s.50-60.

McGuire, G.,Tugemanny, B. & Civarioz, G. (2013) *There is no 16-ClueSudoku: Solving the Sudoku Minimum Number of Clues Problem via Hitting Set Enumeration.* arXiv:1201.0749v2 [cs.DS] 1 Sep 2013.

Morrison, R., & De Jong, K. (2002). *Measurement of Population Diversity.* Lecture Notes In Computer Science, 31-41.

Okagbue, H. I., Omogbadegun, Z. O., Olajide, F. A., & Opanuga, A. A. (2015). *On some suggested applications of sudoku in information systems security.* Asian Journal of Information Technology, Volume 14,

Issue 4, 2015, s.117-121.

Preux P. & Talbi, E-G. (2009) *Assessing the Evolutionary Algorithm Paradigm to Solve Hard Problem.* In Constraints Processing, Workshop on Studying and Solving Really Hard Problems, Marseille, 1995.

Rao, R.V., Savsani, V.J., Vakharia, D.P., 2011. *Teaching-learning-based optimization: A novel method for constrained mechanical design optimization problems*. Computer-Aided Design 43 (3), 303–315.

Redding, J., Schreiver, J., Shrum, C., Lauf, A. & Yampolskiy, R. (2015). *Solving NP-hard number matrix games with Wisdom of Artificial Crowds.* 2015 Computer Games: AI, Animation, Mobile, Multimedia, Educational and Serious Games (CGAMES) doi: 10.1109/CGames.2015.7272959.

Segura, C., Pena, S., Rionda, S. & Aguirre, A. (2016). *The importance of diversity in the application of evolutionary algorithms to the Sudoku problem.* 2016 IEEE Congress on Evolutionary Computation (CEC).

Smit, S., Szláavik, Z., & Eiben, A. (2011). *Population diversity index*. Proceedings Of The 13Th Annual Conference Companion On Genetic And Evolutionary Computation - GECCO '11.

Yato, T. & Seta, T. *Complexity and Completeness of Finding Another Solution and Its Application to Puzzles*. (2003). IEICE Transactions on Fundamentals of Electronics, Communications and Computer Sciences, E86-A (5), s. 1052-1060.

Yue, T. W., & Lee, Z. (2006). *Sudoku solver by q'tron neural networks*. Huang D-S, Li K, Irwin GW (eds) Intelligent computing, LNCS, vol 4113. Springer, Berlin s. 943–952

Wang, Z., Yasuda, T. & Ohkura, K. (2015). *An evolutionary approach to sudoku puzzles with filtered mutations.* 2015 IEEE Congress on Evolutionary Computation (CEC).

Williams, D. J., Jones, S. K., Roach, P. A. & Perkins, S. (2011). Blocking intercalates in Sudoku erasure correcting codes. IAENG International Journal of Computer Science, Volume 38, Issue 3, 2011, s. 183-191.

Wilcoxon, F. (1945). *Individual comparisons by ranking methods.* Biometrics Bulletin. 1 (6): 80–83.

Appendix A Sudokupussel

Sudoku 1

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

Sudoku 2

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

Sudoku 3

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

Sudoku 4

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

Sudoku 5

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

Appendix B LBSA-baserad konstruktor

```
//* Initialize the RNG
//* Read the Sudoku puzzle
//* Read the indices that cannot be changed.
LbsaII::LbsaII(std::vector<int> &sudoku, std::vector<int> &fixedNumbers)
{
	std::random_device rd;
	std::mt19937 gen(rd());
	mRandomEngine = gen;

	mSudoku = sudoku;
	mFixedNumbers = fixedNumbers;
};
```

Appendix C LBSA-baserad *Crossover*-operator

```
//Takes the trial population and a vector of all changeable positions in the Sudoku puzzle as input.
//Each individual in the trial positions has a chance to replace a random amount of positions with the
// value off a individual from the current population.
void LbsaII::crossover(std::vector<Chromosome*> &trial, std::vector<int> &u)

{
     //Create a two dimensional vector and fill every positions with the value 1.
     std::vector<std::vector<int>> crossoverMap(trial.size(), std::vector<int>(mSudoku.size(), 1));
     std::vector<int> changeable;

     //Set random indices in the crossoverMap to zero. The random indices corresponds to a changeable
     position in  the Sudoku puzzle.
     if (randomDouble(0, 1) < randomDouble(0, 1))
     {
         for (int i = 0; i < trial.size(); i++)
         {
             //The variable last determines the maximum amount of position that can be changed.
             //mSettings.mixrate is a constant set at the start of the attempt to solve the Sudoku, its value
             // is determined by the current setting, mSettings.probDim is the number of positions that can
             // be changed in the Sudoku puzzle.
             int last = std::ceil(mSettings.mixrate * randomDouble(0, 1) * mSettings.probDim);

             //u contains indices of the changeable positions of the Sudoku arrays
             changeable = u;
             std::shuffle(changeable.begin(), changeable.end(), mRandomEngine);

             for (int j = 0; j < last; j++)
             {
                 if (changeable.size() != 0)
                 {
                      crossoverMap[i][changeable.back()] = 0;
                      changeable.pop_back();
                 }
             }
         }
     }
     else
     {
         //Set a random position in the crossoverMap to zero.
         for (int i = 0; i < trial.size(); i++)
         {
             crossoverMap[i][randomInt(0, mSudoku.size() - 1)] = 0;
         }
     }

     //Iterate over all individuals in the trial population.
     for (int i = 0; i < trial.size(); i++)
     {

         //Iterate over all positions in the current individuals Sudoku puzzle.
         for (int j = 0; j < mSudoku.size(); j++)
         {

             //If the index of the crossover map for the current individuals (one in the trial population and
             // one in the current population), has the value '1' the value at the index from the individual
             // in the current population should be written in the same index in the individual of the trial
             // population. In order to maintain the condition that the numbers 1-9 should always exist in
             // each subarray, the current value at the index of the indivdual from the trial population must
             // be swapped to the index currently holding the value to be written.
             if (crossoverMap[i][j] == 1 && mFixedNumbers[j] != 1)
             {
                 trial[i]->setSwap(j, mPopulation[i]->getValue(j));
             }
         }
     }
};
```

Appendix D LBSA-baserad mutationsoperator

```
//Takes the trial population, the index of the best individual (best fitness value) and the worst individual
// (worst fitness value).
//mSettings.fMin, mSettings.fMax minimum and maximum numbers of regions allowed to be changed, used to
// make sure individuals receive information from all parts of the equations 2.3.2 and 2.3.3 (in the report).
void LbsaII::mutation(std::vector<Chromosome*> &trial, int best, int worst)
{
    for (unsigned int i = 0; i < trial.size(); i++)
    {

        std::vector<int> regions = { 1, 2, 3, 4, 5, 6, 7, 8, 9 };
        std::shuffle(regions.begin(), regions.end(), mRandomEngine);

        if (randomDouble(0, 1) < randomDouble(0,1))
        {

                int x = randomInt(mSettings.fMin, mSettings.fMax);
                for (int j = 0; j < x; j++)
                {
                    trial[i]->setRegion(regions.back(), mPopulation[i]->getRegion(regions.back()));
                    regions.pop_back();
                }

                int y = 0;
                if (mSettings.fMaxOld - x < mSettings.fMinOld)
                    y = mSettings.fMinOld;
                else
                    y = mSettings.fMaxOld - x;

                y = randomInt(mSettings.fMinOld, y);
                for (int j = 0; j < y; j++)
                {
                    if (regions.size() != 0)
                    {
                          trial[i]->setRegion(regions.back(), mOldPopulation[i]->getRegion(regions.back()));
                          regions.pop_back();
                    }
                }

                int z = 0;
                if (mSettings.fMaxBest - x - y < mSettings.fMinBest)
                    z = mSettings.fMinBest;
                else
                    z = mSettings.fMaxBest;

                z = randomInt(mSettings.fMinBest, z);
                for (int j = 0; j < z; j++)

                {
                    if (regions.size() != 0)
                    {
                          trial[i]->setRegion(regions.back(), mPopulation[best]->getRegion(regions.back()));
                          regions.pop_back();
                    }
                }

        }
        else

        {
            int k = randomInt(0, mPopulation.size() - 1);
            while (k == i) k = randomInt(0, mPopulation.size() - 1);

                    int x = randomInt(mSettings.fMin, mSettings.fMax);
                    for (int j = 0; j < x; j++)
                    {
                          trial[i]->setRegion(regions.back(), mPopulation[k]->getRegion(regions.back()));
                          regions.pop_back();
                    }

            int y = 0;
            int tmpMin = mSettings.fMinBest;
```

```
            if (mSettings.fMaxBest < mSettings.fMinBest)
                y = mSettings.fMinBest;
            else
                y = mSettings.fMaxBest;

            y = randomInt(mSettings.fMinBest, y);
            for (int j = 0; j < y; j++)
            {
                if (regions.size() != 0)
                {
                    trial[i]->setRegion(regions.back(), mPopulation[best]->getRegion(regions.back()));
                    regions.pop_back();
                }
            }

            int z = 0;
            if (mSettings.fMax - x - y < mSettings.fMin)
                z = mSettings.fMin;
            else
                z = mSettings.fMax - x - y;

            z = randomInt(mSettings.fMin, z);
            for (int j = 0; j < z; j++)
            {
                if (regions.size() != 0)
                {
                    trial[i]->setRegion(regions.back(), mPopulation[i]->getRegion(regions.back()));
                    regions.pop_back();
                }
            }

            int v = 0;
            if (mSettings.fMax - x - y - z < mSettings.fMin)
                v = mSettings.fMin;
            else
                v = mSettings.fMax - x - y - z;

            v = randomInt(mSettings.fMinWorst, v);
            for (int j = 0; j < v; j++)

            {
                if (regions.size() != 0)
                {
                    trial[i]->setRegion(regions.back(), mPopulation[worst]->getRegion(regions.back()));
                    regions.pop_back();
                }
            }
        }
    }
};
```

Appendix E LBSA-baserad sökning

```
std::vector<Chromosome*> LbsaII::solve()
{
      int fixed = 0;
      for (int i = 0; i < mFixedNumbers.size(); i++)
      {
          if (mFixedNumbers[i] == 1)
              fixed++;
      }

      mSettings.probDim = mSudoku.size() - fixed;
      mSettings.popSize = 30;
      int bestCount = 0;
      int generations = 0;
      int maxgen = 8000;

      //u holds indices to changeable positions in mSudoku, used in crossover()
      std::vector<int> u = GeneticFunctions::getChangeablePos(mFixedNumbers);

      //1. Initialize population
      //Create population
      mPopulation = GeneticFunctions::createPopulation(mSudoku, mFixedNumbers, mSettings.popSize);
      //Fill the Sudoku puzzel of every individual with random numbers (1-9).
      GeneticFunctions::fillSudoku(mPopulation, mRandomEngine);
      //Create population
      mOldPopulation = GeneticFunctions::createPopulation(mSudoku, mFixedNumbers, mSettings.popSize);
      //Fill the Sudoku puzzel of every individual with random numbers (1-9).
      GeneticFunctions::fillSudoku(mOldPopulation, mRandomEngine);

      GeneticFunctions::calcFitness(mPopulation);
      //Sort ascending based on fitness, best individual will be at position zero.
      GeneticFunctions::sortPopulation(mPopulation);

      int best = 0;
      int worst = 0;
      std::vector<Chromosome*> trial;

      while (mPopulation[best]->getFitness() != 0 && generations <= maxgen)
      {
          //Setting can be varied based on Sudoku puzzle, for example an easy puzzle might use
          // initialize(”easy”);
          // initialize(”settings”); used for puzzles where the standard settings does not work well.
          initialize("settings");

          //2. Selection 1
          if (randomDouble(0, 1) < randomDouble(0, 1) && generations > 20)
          {
              //Cleaning up pointers
              for (int i = 1; i < mOldPopulation.size(); i++)
              {
                  bool inPop = false;
                  for (int j = 0; j < mPopulation.size(); j++)
                  {
                       if (mOldPopulation[i] == mPopulation[j])
                           inPop = true;
                  }

                  if (!inPop)
                       delete mOldPopulation[i];
              }

              mOldPopulation = mPopulation;
          }

          std::shuffle(mOldPopulation.begin(), mOldPopulation.end(), mRandomEngine);

          //3. Generate trial population
          trial = GeneticFunctions::createPopulation(mSudoku, mFixedNumbers, mSettings.popSize);

          //4. Mutation
          mutation(trial, best, worst);
```

```
        //5. Crossover
        crossover(trial, u);

        GeneticFunctions::calcFitness(trial);
        GeneticFunctions::sortPopulation(trial);

        //6. Replacement 2
        for (int i = 0; i < trial.size(); i++)
        {
            if (trial[i]->getFitness() < mPopulation[i]->getFitness())
            {
                Chromosome* tmp = mPopulation[i];
                mPopulation[i] = trial[i];
                trial[i] = tmp;
            }
        }

        std::cout << "Generation: " << generations << std::endl;
        std::cout << "Best Chromosome: " << mPopulation[best]->getFitness() << std::endl;

        //Pointer clean up
        for (int i = 0; i < trial.size(); i++)
        {
            bool inPop = false;
            for (int j = 0; j < mOldPopulation.size(); j++)
            {
                if (trial[i] == mOldPopulation[j])
                    inPop = true;
            }

            if (!inPop)
                delete trial[i];
        }

        generations++;
    }
};
```

Appendix F Randomiseringsfunktion

```
double LbsaII::randomDouble(double min, double max)
{
	std::uniform_real_distribution<> range(min, max);
	return (double)range(mRandomEngine);
};
```