Visualisering og sammenligning af populationsbaserede randomiserede søgeheuristikker

af
Mikkel Dyrstrøm Olin
s103450

Danmarks Tekniske Universitet
DTU Compute
Bachelorprojekt
31. januar 2014

Abstract

This report is one of two products resulting from the work on the problem *"Visualizing and comparing population-based randomized search heuristics"* during the bachelor project. The second product is a program which was designed as a framework to run and test . The report describes the algorithms, fitness functions and selection mechanisms that have been worked with from a theoretical perspective, how this theory have been implemented and how the program is constructed.

Experiments are described and conducted in relation to the theoretical knowledge to compare practical and theoretical data. The results of the experiments showed a good connection between the two kinds of data.

Indhold

1 Indledning

Dette projekt tager fat på emnet *"Visualisering og sammenligning af populationsbaserede randomiserede søgeheuristikker"* som et selvvalgt bachelorprojekt. Arbejdet på projektet er forgået i efteråret 2013 og januar 2014 under vejledning af lektor ved DTU Carsten Witt. Bagrunden for at vælge dette projekt ligger i tidligere arbejde med naturinspirerede algoritmer i kurset 02122 Fagprojekt - Bachelor i Softwareteknologi. I projektet er der fokuseret på algoritmen $(\mu + 1)$EA, samt fire fitness funktioner og selektionsmekanismer.

Produktet af projektet var todelt, idet denne rapport samt et program skulle udarbejdes. Programmet blev skrevet i `Java`.

I reporten gøres der først rede for teorien bag algoritme, fitness funktioner og selektioner. Herefter vil det blive forklaret hvordan denne teori er blevet implementeret i programmet og hvordan rammen omkring er blevet opbygget. Dele af programmet er blevet lånt fra det program som blev udarbejdet i Fagprojektet. Til sidst i rapporten er der en beskrivelse af de eksperimenter der er blevet udført for at undersøge dele af teorien. For hver af disse eksperimenter er der blevet opstillet en hypotese som bunder i delvis teori og gennem samarbejde med Carsten Witt.

I projektet er der blevet arbejdet med en fitness funktion, der i denne rapport er benævnt PreSuf, defineret af Carsten Witt [2], som aldrig før er blevet implementeret og analyseret i praksis.

Artiklen af Carsten Witt analyserer kørertider af $(\mu + 1)$EA på fire fitness funktioner afhængigt af μ og n. Disse fire fitness funktioner er de samme som også bliver beskrevet i denne rapport. Der vil gennem rapporten være gentagende referencer til Carsten Witt's artikel.

2 Teori

2.1 Algoritmen (μ+1)EA

I dette projekt er der arbejdet med $(\mu + 1)$ Evolutionary Algorithm der, som navnet antyder, er inspireret af evolutionær udvikling i naturen. Den bygger på tilfældig mutering af individer i en population indeholdende μ indvider og beslutningen om at lade indvider opleve evolutionen bygger udelukkende på om de er bedre end andre .

Den generelle form af algoritmen er (μ+λ)EA, som også giver anledning til algoritmerne (1+λ)EA og (1+1)EA. Den nedenstående algoritme er ikke en konkret algoritme, men en generel "skabelon"for de fire EA algoritmer:

Algorithm 1 (μ+λ)EA

1: Sæt $t = 0$
2: Generer uniformt tilfældigt individer $x_i \in \{0,1\}^n$ for $1 \leq j \leq \mu$ til populationen X_t
3: **while** true **do**
4: Udvælg λ individer til mængden L fra populationen X_t.
5: Muter individerne i L til L'.
6: Udvælg de μ bedste individer fra $L' \cup X_t$ til X_{t+1}
7: $t = t + 1$
8: **end while**

Her betegner variablen t tiden således, at en population X_t er populationen ved tidspunktet t.

Som algoritmen er skrevet op her så afsluttes den ikke. Dette gøres ved at inføre en *fitness funktion* (se 2.2), som giver en måde at vurderer hvert individ, samt et mål for populationen. Algoritmen stoppes når eet individ har en fitness, der er tilstrækkelig stor. Udvælgelsen af indvider fra populationen sker via en selektion (se 2.3), hvor der for det meste ved udvælgelsen af individer til $t+1$ benyttes Truncation selection.

Muteringen af et individ x_j kan gøres på forskellige måder, men sker oftest ved at flippe hver bit med sandsynlighed $1/n$ for bitstrenge med længden n. Dette betyder, at der i gennemsnit bliver flippet eet bit pr. individ. Denne lave mutationsrate gør, at også små ændringer i indviderne er sandsynlige nok.

Algoritmen (μ+1)EA opbygges på lignende vis, hvor der kun udvælges eet individ fra population til at blive muteret.

Algorithm 2 (μ+1)EA

1: Sæt $t = 0$
2: Generer uniformt tilfældigt individer $x_j \in \{0,1\}^n$ for $1 \leq j \leq \mu$ til populationen X_t
3: **while** true **do**
4: Udvælg eet individ x til mængden fra populationen X_t.
5: Muter x til x'.
6: Udvælg de μ bedste individer fra $x' \cup X_t$ til X_{t+1}
7: $t = t + 1$
8: **end while**

2.2 Fitness Funktioner

For at give algoritmen et mål benyttes en fitness funktion, som udregner en værdi for hvor langt bitstregene er fra målet, der som regel er et optimum. Dette gør at fitness funktionen fungere som en guide for hvordan populationen udvikles. I dette projekt er det kun een af individerne, som skal finde optimum, men det kunne også dreje sig om hele populationen. Det er i programmet muligt at vælge fire forskellige fitness funktioner: OneMax, LeadingOnes, SPC og PreSuf som beskrevet i [2]. Disse fitness funktioner har stor interesse i den teoretiske litteratur (Auger & Doerr - *"Theory of RSH"*), hvilket gør dem interesante at undersøge i praksis.

2.2.1 OneMax

Denne fitness funktion tæller det totale antal af 1'taller i bitstrengene:

$$OneMax(x) = \sum_{i=1}^{n} x_i \tag{1}$$

hvor n er længden af bitstrengen.

Den teoretiske kørertid for OneMax på $(\mu + 1)$EA er jf. [2] $O(\mu n + n \cdot \log n)$.

2.2.2 Leading Ones

Denne fitness funktion tæller antallet af 1'erer i begyndelsen af bitstrengene:

$$LeadingOnes(x) = \sum_{i=1}^{n} \prod_{j=1}^{i} x_j \tag{2}$$

hvor n er længden af bitstrengen.

Den teoretiske kørertid for Leading Ones på $(\mu + 1)$EA er jf. [2] $\Theta(\mu n \log n + n^2)$.

2.2.3 Short Path with Constant fitness - SPC

Denne fitness funktion omdanner bitstrengene til kun at bestå af 0'er. Derefter vil funktionen opbygge bitstrengene til kun at bestå af 1'er på samme måde som Leading Ones, dog under påvirkining af konstant fitness værdi. Dette betyder at bitstrengene ikke bliver guidet mod optimum som i OneMax og Leading Ones, men i stedet gennem en såkaldt "random walk"på et plateau. Dette bivirker, at kørertiden bliver meget stor.

$$SPC(x) = \begin{cases} n+1 & \text{hviss } x = 1^i 0^{n-i} \text{ for } i < n \\ 2n & \text{hviss } x = 1^n \\ n - OneMax(x) & \text{ellers} \end{cases} \tag{3}$$

Den teoretiske køretider for SPC på $(\mu + 1)$EA er jf. [2] $\Omega(\mu \cdot n^3/log\mu)$ og $O(\mu \cdot n^3)$.

2.2.4 Prefix Ones og Leading Suffix Ones - PreSuf

Denne fitness funktion blev skabt med det ene formål at undersøge fordelene ved algoritmen $(\mu + 1)$EA ved eksistensen af et globalt og lokalt optimum i søgerummet. Dette skyldes, at populationen for $(\mu + 1)$EA hjælper med at udforske søgerummet bedre. Det kan teoretisk bevises, at $(1 + 1)$EA kun finder det lokale optimum, hvorimod $(\mu + 1)$EA vil finde det globale optimum i langt de fleste tilfælde for lange bitstrenge. Derfor ville et eksperiment være oplagt at udfører for at tjekke dette i praksis.

Hvert individ i populationen opdeles i en prefix del, med længde m, og en suffix del, med længde $\ell = \sqrt{n}/45$.

For et individ x defineres hjælpefunktionerne:

$$PO(x) = \sum_{i=1}^{m} x_i \tag{4}$$

hvor PO står for Prefix Ones og

$$LSO(x) = \sum_{i=0}^{\ell-1} \prod_{j=0}^{i} x_{m+1+j} \tag{5}$$

hvor LSO står for Leading Suffix Ones. Det ses, at hjælpefunktionerne udregner det samme som henholdsvis OneMax og Leading Ones.

Derudover defineres

$$b = 2 \cdot m/3 + \lceil \sqrt{n}/(55 \cdot \log^2 n) \rceil \tag{6}$$

som også benyttes i udregningen af indvidernes fitness.

Derved kan følgende fitness funktion opstilles:

$$PreSuf(x) = \begin{cases} PO(x) + n^2 \cdot LSO(x) & \text{hviss } PO(x) \leq \frac{2m}{3} \\ n^2\ell - n \cdot |PO(x) - b| + LSO(x) - \ell - 1 & \text{ellers} \end{cases} \tag{7}$$

Den indvilklede definition skyldes, at beviserne i [2] har behov for omhyggeligt valgte konstanter (45, 55, etc.)

Det globale optimum for denne fitness funktion findes når der for eet indvid x i populationen gælder $PO(x) = 2m/3$ og $LSO(x) = \ell$. Dette betyder, at et indvid kun kan finde det lokale optimum hviss $LSO(x)$ ikke er blevet optimeret før $PO(x) = 2m/3$.

2.3 Selektioner

I dette projekt er fire selektions mekanismer blevet implementeret og analyseret: Unifomly Random, Fitness Proportional, Tournament og Truncation.

Hvilke individer der udvælges fra populationen kan have indflydelse på hvor mange iterationer der skal bruges før algoritmen finder optimum, I_O. Derudover kan selektionen i visse tilfælde være afgørende for hvorvidt det er muligt at finde optimum pga. det såkaldte *selektionspres*. For simple fitness funktioner med kun eet optimum, såsom OneMax og Leading Ones, er I_O og selektionespresset som regel hinandens modsætninger, så en selektion med højt selektionspres har lav kørertid.

Antallet af individer, der udvælges, afgøres af værdien af λ fra algoritmen, som i dette projekt er 1.

2.3.1 Unifomly Random

selektionen udvælger uniformt tilfældigt eet individ i populationen. Dette betyder at presset for at vælge eet individ er meget lavt.

2.3.2 Fitness Proportional - Roulette Wheel

Fitness Proportional selektion dækker over en mængde af selektioner såsom Roulette Wheel selektion og Stochastic Universal Sampling [1]. Disse selektioner har det tilfældes, at alle individer i populationen har en chance for at blive valgt, dog med en favorittisering af de individer, der relativt til andre individer i populationen, har en bedre fitness. I dette projekt er versionen Roulette Wheel blevet brugt. Dette vil sige at sandsynligeheden for at et individ x bliver valgt er $\frac{f(x)}{\sum_{i=1}^{\mu} f(x_i)}$ for $f(x)$ værende fitness værdien for individ x.

2.3.3 Tournament

Denne selektion fungerer langt hen ad vejen på samme måde som Roulette Wheel selektionen, med den undtaglese, at det kun er på en delmængde af populationen i stedet for hele populationen. Denne delmængde udvælges uniformt tilfældigt. Grunden til, at selektionen kaldes Tournament er, fordi man også kan se på det som en delmængde, hvor individerne "kæmper"mod

hinanden i en turnering for at blive udvalgt til at mutere. Det er dog stadig problematisk at kalde selektions mekanismen Tournament da hver turnering er baseret på tilfældigheder.

2.3.4 Truncation

Denne selektion udvælger det individ i populationen, som har den relativt bedste fitness. Dette betyder at denne selektion får algoritmen $(\mu + 1)$EA til at opfører sig næsten som algoritmen $(1 + 1)$EA. Forskellen er, at ved genereringen af den første population bliver chancen for at generere en godt indvid i forhold til fitness forbedres. Dette kan medfører et fald i kørertiden ved store værdier for μ. Derudover vil et indvid, som er opstået som en mutation af et individ i populationen, have samme sandsynlighed for at blive optaget i populationen som sin "forælder". Fordi denne selektion kun går efter det bedste indvid, så har den et meget stort selektionspres.

3 Implementering af teoretiske modeller

3.1 Population

For at gøre det lettere at overfører værdier tilknyttet en population blev klassen `Population`, som extender `ArrayList<boolean[]>` oprettet. Denne klasse indeholder også funktionen `generate()`, som sørger for at generere bitstrengene i populationen.

Som nævnt er hver bit i programmet representeret af en boolean værdi, hvor true = 1 og false = 0. Dog vil bitværdier senere stadig betegnes som 1'er og 0'er.

3.2 $(\mu+1)$EA

Algoritmen er bygget op omkring Algoritme 2 i Sektion 2.1, hvor der først udvælges et individ fra populationen. Dernæst muteres det valgte indvid, lægges til populationen og til sidst findes det individ i populationen som har den relativt dårligste fitness. Det er også har at det tjekkes om PreSuf

Det skal i øvrigt nævnes at den teknik der bruges for at finde det dårligste indvid, som skal fjernes, i populationen er den samme som Truncation selektion.

3.3 Fitness Funktioner

Fælles for alle fitness funktionerne er, at de nedarver fra interfacet `FitnessFunction`, som sørger for, at hver fitness funktion indeholder følgende funktioner:

- `double f (boolean[] p)`, til at udregne fitness for een bitstreng.
- `String getName()`, til at fortælle hvilken fitness funktion der er i brug.
- `String theoreticalRuntime(int my, int n, String s)`, til at udregne teoretisk kørertid fra littraturen.
- `void Initialize(int n, double c)`, til at initialisere.
- `boolean isOptimized(Population p)`, til at finde ud af om der er een bitstreng i populationen, som er optimeret i forhold til fitness funktionens mål.

Fitness funktionernes `f` funktion er lagt i Appendix A. Der vil i hvert afsnit være henvisning til hvor i Appendix A den tilhørende `f` funktion er.

3.3.1 OneMax

`f` funktionen kan findes i appendix A.1.1.

Funktionen gennemgår hver enkelt bit i en bitstreng og lægger 1 til bitstrengens fitness hver gang der fremkommer et 1 i strengen. Når hele bitstrengen er kørt igennem bliver resultatet returneret.

3.3.2 Leading Ones

`f` funktionen kan findes i appendix A.1.2

Funktionen er opbygget på samme måde som OneMax funktionen, dog med den forskel, at ved fremkomsten af et 0 i bitstrengen bliver fitness værdien returneret.

3.3.3 SPC

`f` funktionen kan findes i appendix A.1.3

I første omgang udregnes OneMax, `total`, og Leading Ones, `front`, værdier for bitstrengen, da disse skal bruges til udregning og case identifikation. Når disse værdier er udregnet går funktionen videre til at finde ud af hvilken case der er opfyldt.

Hvis værdien af `front` er den samme som bitstrengens længde så er optimum fundet og funktionen udregneer og returnere den tilhørerne fitness.

Hvis `front` er lig `total` og `front` desuden er mindre end bitstrengens længde så returneres bitstrengens længde adderet med 1.

Er ingen af de ovenstående betingelser opfyldt så returneres bitstrengslængden minus `total`.

3.3.4 PreSuf

`f` funktionen kan findes i appendix A.1.4

Overgangen fra teori til praksis har medført simplificeringer i modellen da det ønskede, teoretiske resultat for denne fitness funktion ikke fremkom ved en nøjagtig teoretisk implementering. Dette skyldes, at størrelsen af n ikke kan vælges tilstrækkelig stor nok i praksis pga. konstanterne (45, 55 etc.). Derfor blev følgende ændringer foretaget:

- $\ell = \sqrt{n}$, som betyder at suffix strengen er længere for lange bitstrenge.
- $b = 3m/4$, som betyder at det lokale optimum flyttes (især pga. visualiseringen).

I starten opdeles individet i en prefix og suffix del. Derved kan de to hjælpefunktioner `int PO(boolean[] b)` og `int LSO(boolean[] b)`, som fungere henholdsvis på samme måde som OneMax og Leading Ones, benyttes. Når dette er gjort bliver værdien `po` (Prefix Ones) holdt sammen med 2m/3 for at finde ud af hvilken case der skal vælges til at udregne fitness.

Det er nu spændende at undersøge hvorvidt den adfærd, der i litteraturen bevises for den oprindelige PreSuf funktion, også kan bekræftes for den simplificerede version.

3.4 Selektioner

Fælles for alle selektionerne er, at de nedarver fra interfacet `Selection`, som sørger for, at hver fitness funktion indeholder følgende funktioner:

- `boolean[] s ()`, til at udvælge een bitstreng fra populationen.
- `String getName()`, til at fortælle hvilken selektion der er i brug.
- `void Initialize(Population p, FitnessFunction f)`, til at initialisere.

Selektionernes `s` funktion er lagt i Appendix A. Der vil i hvert afsnit være henvisning til hvor i Appendix A den tilhørende `s` funktion er.

3.4.1 Unifomly Random

`s` funktionen kan findes i appendix A.2.1

Den tilfældige udvælgelse sker ved at benytte `Java`'s indbyggede `Math.Random()` funktion. Da denne funktion genererer en tilfældig værdi mellem 0 og 1 skal den tilfældige værdi multipliceres med antallet af bitstrenge. Dette giver position i population og den bitstreng, som befinder sig på denne position bliver returneret.

3.4.2 Fitness Proportional

`s` funktionen kan findes i appendix A.2.2

Før der kan udvælges en bitstreng fra populationen skal hver enkelt bitstreng have udregnet og sammenlignet dens fitness ud fra den valgte fitness funktion. Dette sker i funktionen `luckyNumberGenerator()`, hvor der udregnes hvor stor en procentdel hver enkel bitstrengs fitness har af populations samlede fitness dvs. $\sum_{i=1}^{\mu} f(x_i)$ for en givet fitness funktion `f`.

Disse værdier ligges ind i et array, således at en bitstrengs position i populationen svarer til bitstrengs værdi i arrayet. For plads m i arrayet summes alle de forgående værdier med fitness procentdelen af bitstreng m i populationen. På den måde opbygges en form for roulette, hvor hver bitstreng representeres af et tal. Dette gør det nemmere når der skal udvælges en bitstreng, da man så kan benytte den indbyggede `Math.Random()` funktion.

For hver værdi i arrayet spørges nu om den tilfældige værdi er mindre eller lig værdierne i arrayet. Første gang dette er opfyldt returneres den representerede bitstreng.

I enden af funktionen er der et syntaktiskt `return` statement da `Java` ellers vil melde, at der var en fejl i koden.

3.4.3 Tournament

`s` funktionen kan findes i appendix A.2.3

Først bliver der tilfældigt udvalgt en delmængde af populationen på $k = \mu/20$ indvider vha. `Math.Random()` funktionen. Grunden til, at $k = \mu/20$ er for det første at sørge for, at delmængden er afhængig af populationens størrelse (i stedet for at være konstant) og dernæst, at delmængden er lille nok.

Herefter sorteres mængden af individer via.`sort()` efter fitness så det individ, der har den bedste fitness er på første plads i arrayet.

Udvælgelsen af det individ, som skal muteres, sker ved først sætte det bedst individ op mod det næstbedste. Dette sker i praksis ved at udregne en sandsynlighed for, at det dårlige individ bliver valgt frem for det bedste. Denne sandsynlighed bliver udregnet via funktionen $cr(x_i) = 0.35^i$, hvor $i = 2...k$. Den konstante værdi 0.35 er valgt ud fra eksperimentering med selektionen og ud fra, at det bedste individ i delmængden bør have de største chancer for at blive valgt til mutering. Når alle individer er blvet kørt igennem returneres det individ som "vandt turneringen".

I stedet for denne metode kunne Tournament opbygges på samme måde som Fitness Proportional selektionen, som kørte på en delmængde af populationen.

3.4.4 Truncation

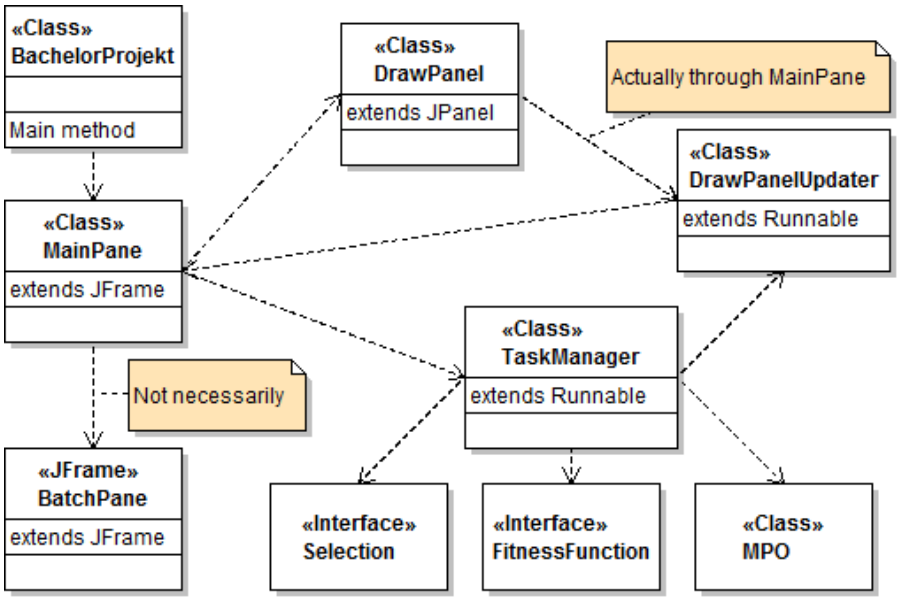
`s` funktionen kan findes i appendix A.2.4

Til at starte med antages, det at den første bitstreng i populationen har den bedste fitness. Herefter køreres hele populationen igennem for at finde ud af om der er bitstrenge, som har skarpt bedre fitness. Hvis sådan en findes forkastes den foregående antagelse og det antages nu, at den bedre bitstreng er den bedste i populationen.

4 Opbygning af program og visualisering

Programmet består af i alt 18 filer hvoraf der allerede er gjort rede for 12 af disse filer i sektion 3. Dette afsnit handler derfor de retsende dele af programmet samt hvordan det hele bindes sammen.

Opbygningen af programmet kan ses i nedenstående klassediagram, hvor pilene mellem klasserne betyder afhængighed mellem dem.



Figur 1

At TaskManager er afhængig af de to interfaces betyder egentlig, at de er afhængige af de klasser, der extender disse interfaces.

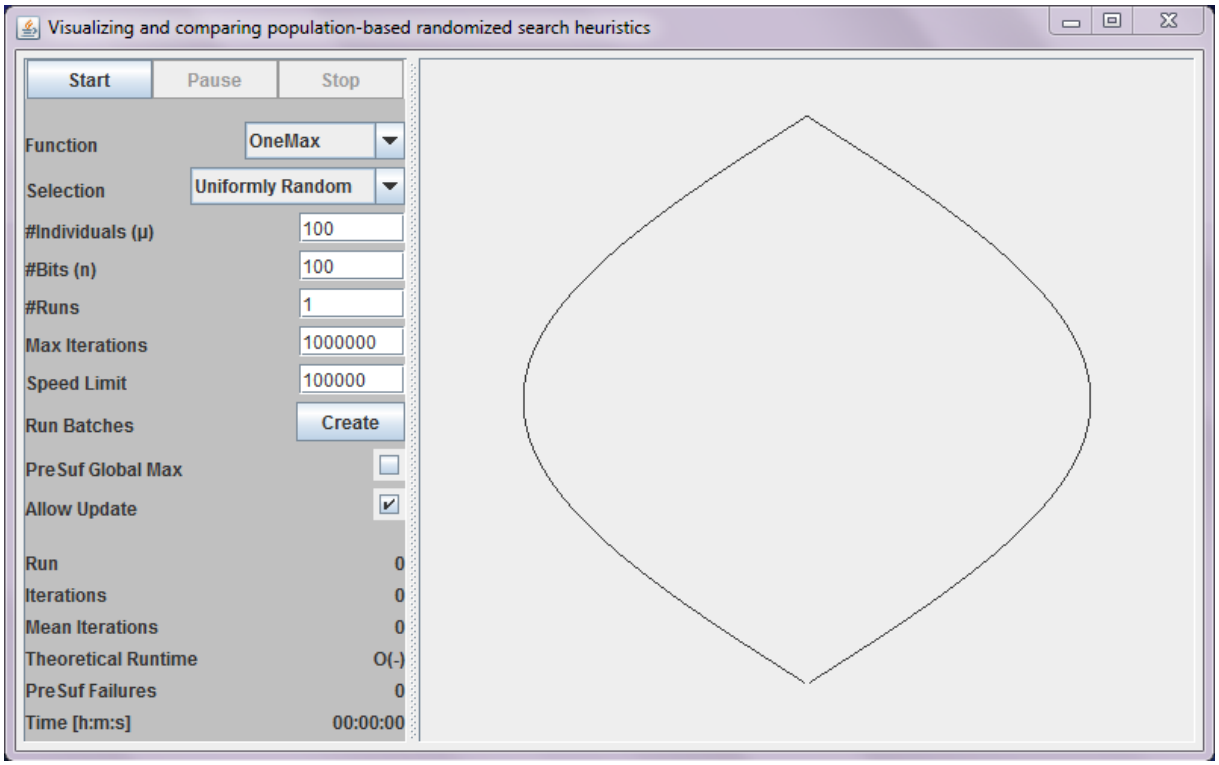
Programmet kan enten få input data fra MainPane eller BatchPane, men ikke fra begge på samme tid. Data fra BatchPane går gennem MainPane for at komme videre, hvilket er forsøgt vist i diagrammet.

Som det sikkert bemærkes i klasse diagrammet så er Population ikke nævnt. Dette skyldes at Population typen bruges af stort set alle klasser i programmet med undtagelse af BachelorProjekt og BatchPane (og selvfølgelig også de to interfaces) og det ville blive for mange linjer at skulle holde øje med.

Det eneste klassen BachelorProjekt gør er at oprette et MainPane for at starte programmet og vil derfor ikke blive nævnt igen.

4.1 MainPane

MainPane står for den primære visuelle del af programmet samt overførsel af data mellem TaskManager, DrawPanel og BatchPane. Det er bygget op som et frame med fire paneler, hvor hvert panel står for hver sin del af visualiseringen.



Figur 2: MainPane med Control (øverst ventre), input (midt ventre), output (nederst venstre) og drawpanel/hypercube (højre)

4.1.1 Control panel

Dette panel indeholder tre knapper: Start, Pause og Stop.

Start knappen sørger for at indhente data fra MainPane input panelet eller BatchPane input tabellen hvis der er et BatchPane åbent. Den data, der bliver hentet, lægges i et `ArrayList<Object[]>`, som videregives til `TaskManager` og herefter bliver `taskManager.start()` kaldt. Hvis teksten på knappen er Resume så bliver `taskManger.start()` blot kaldt.

Pause knappen sætter teksten på Start knappen til "Resume"og kalder `taskManager.pause()`.

Stop knappen kalder `taskManager.stop()`.

4.1.2 Input panel

Her kan brugeren vælge og/eller indtaste information om een batch

Function En `JComboBox` som gør det muligt for brugeren at vælge en fitness funktion.

Selection En `JComboBox` som gør det muligt for brugeren at vælge en selektions mekanisme.

#Individuals Et `JTextField` som gør det muligt for brugeren at vælge antal individer i populationen.

#Bits Et `JTextField` som gør det muligt for brugeren at vælge antal bits pr. individ.

#Runs Et `JTextField` som gør det muligt for brugeren at vælge antal kørsler.

Max Iterations Et `JTextField` som gør det muligt for brugeren at vælge det maximale antal iterationer pr. kørsel.

Speed Limit En `JTextField` til at sætte hvor mange udregninger der må udføres pr. millisekund

Run Batches En `JButton` som åbner BatchPane 4.2.

Derudover er der to specielle input felter, som begge er `JCheckBox`.

PreSuf Global Max som styrer om det globale maximum for en kørslen med PreSuf bliver vist.

Allow Update som styrer hvorvidt drawpanel bliver opdateret. Dette gør, at der ikke bliver brugt nær så meget processorkraft.

4.1.3 Output panel

Her vises løbende information om populationens udvikling i den enkelte batchkørsel. I panelet er der 6 output felter som alle er `JLabel`'s:

Run viser hvilken kørsel i den pågældende batch programmet er i.

Iterations viser hvilken iteration i den pågældende kørsel programmet er i.

Mean Iteration viser det gennemsnitlige antal itrationer der er blevet brugt i den pågældende kørsel.

Theoretical Runtime viser den teoretiske kørertid (hvis sådan en er kendt) afhængig a fitness funktion og slektions mekanisme.

PreSuf Failures viser antallet af gange det lokale maximum er fundet i den pågælde batch for PreSuf.

Time viser hvor lang tid en kørsel har kørt.

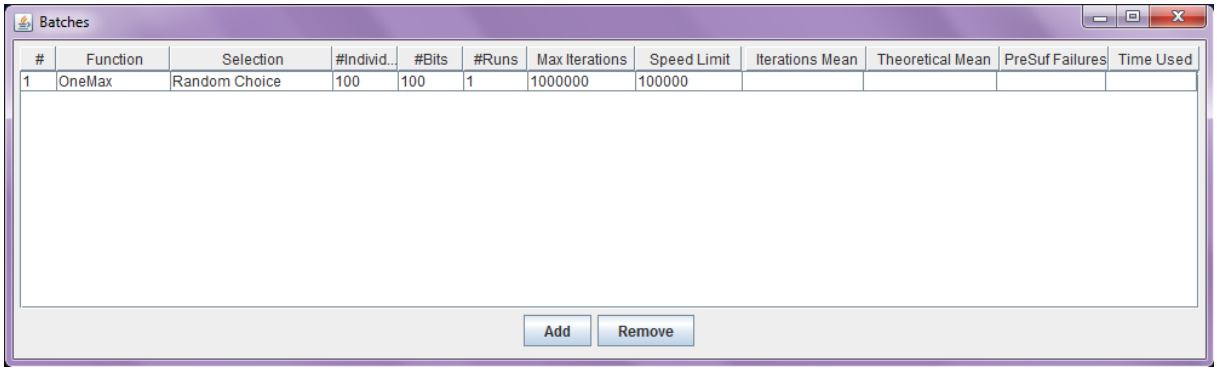
4.1.4 DrawPanel

DrawPanel er en klasse som extender `JPanel` og viser en såkaldt Hypercube, som er en visuel representation af antallet af 1'er og 0'er i bitstrengen, samt vægten af dem. Vertikalt vises antallet af 1'er og 0'er (1'er øverst og 0'er nederst). Horisontalt representeres vægten af 1'er i strengen, således at i øverste venstre del betyder, at der er relativt mange 1-taller i første halvdel af strengen sammenlignet med den anden halvdel. I hypercuben bliver individer i populationen vist som prikker med forskellige farve (afhængig af populationen størrelse), som bevæger sig rundt i takt med, at de bliver bearbejdet.

4.2 BatchPane

Dette vindue giver brugeren mulighed at sætte programmet til at kører flere forskellige batches. Denne mulighed gør det lettere at kører mange batches uden at skulle indtaste nye værdier hver gang en batch er færdig.

Dette frame er bygget op således at det indeholder knapperne Add og Remove, samt tre tabeller inde i eet ScrollPanel: Rowcount, input, og output



Figur 3: BatchPane med rowcount (ventre), input (midt), output (højre)

Rowcount viser nummerering af rækker for hver batch.

Input tabellen har felter ligesom input felterne i MainPane input panelet, med undtagelse af `JCheckBox`'ene. Det er muligt at indsætte nye rækker ved at trykke på Add. Som default vil Add kopiere den data som brugeren har skrevet i nederste række som data i den nye række. For at fjerne data fra tabellen skal brugeren makere nummererings ud for de rækker der ønskes fjernet og trykke på Remove knappen.

Output tabellen viser output data for det gennemsnitlige antal iterationer, den teoretiske kørertid, antal lokale optimum fundet for PreSuf og tiden det tog at kører batchen. Denne tabel opdateres via TaskManager når en batch er færdigkørt.

4.3 TaskManager

Denne klasse er den overordnede del i kørslerne. Det er her parameter data (fra MainPane) bliver identificeret og initialiseret så det er brugbart til kørsel af batches. Dette gælder især for fitness funktioner og selektioner da disse gives som strenge. Derfor er der blevet defineret to hjælpefunktioner som kan finde ud af hvilken fitness funktion og selektion brugeren har valgt.

`TaskManager` extender Runnable således at den kan kørers i en tråd der gør det muligt at starte, pause og stoppe en kørsel. Start og Stop sker vha. de indbyggede metoder og tråde `start()` og `interrupt()` hvorimod Pause styres via en boolean variabel. Dette sker ved, at boolean varablens værdi tjekkes under kørslen og hvis den er true så bliver kørslen "fanget"i et while-loop hvor den indebyggede funktion `wait()` kaldes.

Hvor mange udregninger der udføres pr. sekund styres også herfra ved at bruge den værdi brugeren har givet for Speed Limit. Styreringen med hvor lang tid tråden skal pauses sker ved at benytte den indbyggede metode `sleep(t)` som får tråden til at "sove"i t millisekunder. For at denne metode kaldes ses der på om antallet af iterationer der erkørt igennem siden tråden sidst sov. Hvis dette antal er størrer end værdien for Speed Limit så bliver `sleep()` kaldt.

4.4 DrawPanelUpdater

`DrawpanelUpdater` og bruges udenlukkende af `TaskManager` til at opdatere det DrawPanel som er i MainPane. Fordi `DrawpanelUpdater` extender Runnable kan `TaskManager` benytte den indbyggede `SwingUtilities.invokeLater` til at kalde `DrawpanelUpdater` så GUI'en kan opdateres.

5 Eksperimenter

De teoretiske kørertider der arbejdes med i dette afsnit er alle fra [2]. Resultaterne af testene er lagt i Appendix B og bearbejdelsen af resultaterne kan findes i Appendix C. Der vil i hvert eksperiment være henvisninger til hvor i hvert appendix de tilhørende resultater og bearbejdelser er. Længden af bitstrengene i testene betegnes som n.

5.1 Køretid med OneMax og Unifomly Random

Den teoretiske øvre grænse for OneMax med Unifomly Random identificeres $O(\mu \cdot n + n \cdot log(n))$ og det vil derfor være oplagt at teste i hvilket interval konstanten fra O ligger i.

5.1.1 Hypotese

Teoretisk set vil antallet af iterationer brugt på at finde optimum nærme sig en konstant værdi i forhold til den teoretiske køretid. Dette betyder at $\frac{f(\mu,n)}{\mu \cdot n + n \cdot log(n)}$ vil ligge stabilt for f værende køretiden af algoritmen $(\mu + 1)EA$, der benytter fitness funktionen OneMax og selektionen Unifomly Random.

5.1.2 Opsætning

Der er blevet kørt test med $\mu = \{1, 10, 20, 30, 40, 50, 60, 70, 80, 90, 100\}$ og
$n = \{10, 20, 30, 40, 50, 60, 70, 80, 90, 100\}$. Antallet af kørsler blev sat til 1000.

5.1.3 Resultater

Resultaterne kan findes i tabel B.1 og behandling af resultater findes i C.1.

I tabel C.1 ses det, at der kun forekommer små svingninger i forholdet mellem den praktiske og teoretiske kørertid for $n \geq 60$ og $\mu \geq 40$. Dette må betyde at konstanten for den teoretiske køretid formentlig ligge i intervallet mellem (for (n, μ) værende et felt i tabel C.1) $(60, 100) = 0,834$ og $(100, 40) = 0,977$.

5.2 Køretid med OneMax og Truncation

5.2.1 Hypotese

Da det altid er det relativt bedste individ som bliver valgt, vil kørertiden blive uafhængig af hvor mange individer, der er i populationen. Man vil defor se at resultaterne vil forholde sig stabilt selvom værdien for μ ændres. Som beskrevet i 2.3.4 vil Truncation selektionen få algoritmen $(\mu + 1)$EA til at opfører sig som $(1 + 1)$EA. Derfor skulle kørertiden nærme sig $O(n \log n)$ som er den teoretiske kørertid for $(\mu + 1)$EA med OneMax.

5.2.2 Opsætning

Der er blevet kørt test med $\mu = \{1, 10, 20, 30, 40, 50, 60, 70, 80, 90, 100\}$ og
$n = \{10, 20, 30, 40, 50, 60, 70, 80, 90, 100\}$. Antallet af kørsler blev sat til 1000.

5.2.3 Resultater

Resultaterne kan findes i tabel B.2.

Det ser ud til, at ved stigende værdi for μ er der et lille fald i kørertiden, hvilket især kan ses ved korte bitstrenge. Grunden til dette kunne være, at sandsynligheden for at generere et godt individ ved initialiseringen stiger, når der genereres mange individer, som forklaret i sektion 2.3.4. For at finde ud af om denne hypotese holder er følgende ekstra subeksperiment blevet kørt for $\mu = 1000$:

$\mu \backslash n$	10	20	30	40	50	60	70	80	90	100
1000	7	90	183	288	391	500	622	748	893	1007

Ud fra disse resultater kan det konkluderes, at ved stigende μ værdi vil køretiden falde, dog i mindre grad for stigende n værdi.

5.3 Sammenligning af Selektioner på Onemax

Som beskrevet i starten af afsnit 2.3 er køretiden og selektionspresset tit omvendt proportionale for simple fitness funktioner. I dette eksperiment vil det blive undersøgt om dette er sandt ud fra viden om selektionspresset for de enkelte selektioner.

5.3.1 Hypotese

Køretiden vil fordele sig således at Unifomly Random har det største gennemsnit i antal brugte iterationer, efterfulgt af Fitness Proportinal, Tournament og til sidst Truncation. Dette vil vise at Unifomly Random og Truncation ligger i hver sin ende af selektionspres skalaen

5.3.2 Opsætning

Resultater for Unifomly Random og Truncation bliver genbrugt fra Eksperiment 1 og 2. Eksperimenterne for Fitness Proportional og Tournament er blevet kørt på samme måde som Eksperiment 1 og 2.

5.3.3 Resultater

Resultaterne kan findes i tabel B.1, B.2, B.3 og B.4.

Det ses at Unifomly Random bruger et markant antal flere iterationer i gennemsnittet end Truncation. Derudover ligger Fitness Proportional tættest på Unifomly Random efterfulgt af Tournament. Disse data påviser hypotesen om at Unifomly Random og Truncation ligger i hver deres ender af selektionspres skalaen.

5.4 Køretid med Leading Ones og Unifomly Random

Den teoretiske køretid for Leading Ones med Unifomly Random identificeres til $\Theta(\mu \cdot n \cdot log(n) + n^2)$ og det vil derfor være oplagt at teste i hvilket interval konstanten ligger i.

5.4.1 Hypotese

Teoretisk set vil antallet af iterationer brugt på at finde optimum nærme sig en konstant værdi i forhold til den teoretiske køretid. Dette betyder at $\frac{f(\mu,n)}{\mu \cdot n \cdot log(n) + n^2}$ vil ligge stabilt for f værende køretiden for algoritmen $(\mu + 1)EA$, der benytter fitness funktionen Leading Ones og selektionen Unifomly Random.

5.4.2 Opsætning

Der er blevet kørt test med $\mu = \{1, 10, 20, 30, 40, 50, 60, 70, 80, 90, 100\}$ og
$n = \{10, 20, 30, 40, 50, 60, 70, 80, 90, 100\}$. Antallet af kørsler blev sat til 1000.

5.4.3 Resultater

Resultaterne kan findes i tabel B.5 og behandling af resultater findes i C.3.

I tabel C.3 ses det, at der kun forekommer små svingninger i forholdet mellem den praktiske og teoretiske kørertid for $n \geq 50$ og $\mu \geq 30$. Dette betyder, at konstanten for den teoretiske køretid formodentlig ligger i intervallet (for (n, μ) værende et felt i tabel C.1) $(50, 100) = 0,319$ og $(100, 30) = 0,469$.

5.5 Køretid med Leading Ones og Truncation

5.5.1 Hypotese

Da det altid er det relativt bedste individ som bliver valgt, vil kørertiden blive uafhængig af hvor mange individer, der er i populationen. Man vil defor se at resultaterne vil forholde sig stabilt

selvom værdien for μ ændres. Som beskrevet i 2.3.4 vil Truncation selektionen få algoritmen $(\mu + 1)$EA til at opfører sig næsten som $(1 + 1)$EA.

5.5.2 Opsætning

Der er blevet kørt test for $\mu = \{1, 10, 20, 30, 40, 50, 60, 70, 80, 90, 100\}$ og
$n = \{10, 20, 30, 40, 50, 60, 70, 80, 90, 100\}$. Antallet af kørsler blev sat til 1000.

5.5.3 Resultater

Resultaterne kan findes i tabel B.6 og behandling af resultater findes i C.4.

Lige som med Eksperiment 2 er der et fald i kørertiden ved stigene μ værdi. Derfor udførers samme subeksperiment med $\mu = 1000$.

$_{\mu}\backslash^{n}$	10	20	30	40	50	60	70	80	90	100
1000	11	215	595	1147	1874	2773	3810	5083	6499	8057

Ud fra disse resultater kan det konkluderes, at ved stigende μ værdi vil køretiden falde, dog i mindre grad for stigende n værdi.

5.6 Køretid med SPC og Unifomly Random

Den teoretiske nedre og øvre kørertider for SPC med selektion Unifomly Random identficeres til henholdsvis $\Omega(\mu \cdot n^3/log\mu)$ og $O(\mu \cdot n^3)$. Hvilken af den nedre eller øvre grænse, der afspejler kørertiden bedst, med hensyn til konstanter, vil blive undersøgt.

5.6.1 Hypotese

Forholdet mellem den øvre og den praktiske kørertid eller den nedre og den praktiske kørertid vil ligge stabilt.

5.6.2 Opsætning

Fordi der her er tale om kørertider, som er bundet af længden af bitstrengene i tredje potens så er kørertiderne utrolig lange. Derfor er testene kørt for $\mu = \{1, 10, 20, 30, 40, 50\}$ og $n = \{10, 20, 30, 40\}$. Antallet af kørsler blev sat til 1000.

5.6.3 Resultater

Resultaterne kan findes i tabel B.7 og behandling af resultater findes i C.5 og C.6.

Umiddelbart virker som om, at forholdet mellem den øvre teoretiske kørertid og den pratiske kørertid forholder sig mest stabilt. Det er dog svært at sige om dette resultat holder for størrer værdier for μ og n, men det svært at undersøge pga. de store kørertider.

5.7 Sammenligning af fejl på PreSuf med Unifomly Random og Truncation

I alle de foregående eksperimenter har formålet været at sammenligne optimal kørertider med en teoretisk kørertid eller forskellige selektioner. Det er imidlertid ikke interessant for denne PreSuf da søgerummet både indeholder et globalt og lokalt optimum. Som beskrevet i afsnit 2.2.4 er det derfor ikke sikkert, at det globale optimum bliver fundet. Dette giver imidlertid anledning til en anden form for eksperiment: Hvor mange gange findes det lokale optimum, frem for det globale, for hver af de fire selektioner?

5.7.1 Hypotese

Pga. Truncation selektionens høje selektionspres så forventes det, at der i kørsler med denne selektion vil opstå flere fejl. Dvs., at algoritmen vil finde det lokale optimum flere gang end ved kørsler med Unifomly Random, som har et lavt selektionspres. Fitness Proportional og Tournament vil befinde sig imellem Unifomly Random og Truncation, da de er en form for blanding af Unifomly Random og Truncation.

5.7.2 Opsætning

Der er blevet kørt test for $\mu = \{1, 10, 20, 30, 40, 50, 60, 70, 80, 90, 100, 1000\}$ og
$n = \{10, 20, 30, 40, 50, 60, 70, 80, 90, 100, 1000\}$. Antallet af kørsler blev sat til 1000.

5.7.3 Resultater

Resultaterne kan findes i tabel B.8, B.9, B.10 og B.11.

Med Truncation findes et markant lavere antal globale optimum i forhold til Unifomly Random, hvilket især kan ses for et stort antal af lange bitstrenge. Dette viser, at pga. presset for at vælge det bedtse individ så får Truncation, algoritmen til ramme forbi det globale optimum. Da Truncation selection får $(\mu + 1)$EA til at opføre sig næsten som $(1 + 1)$EA, så ser det ud til at bevise hypotesen. Dette stemmer godt overens med beviserne fremsat af Witt [2], som siger, at $(1 + 1)$EA næsten aldrig vil finder det globale optimum inden for polynomial tid. Det ville dog kræve et eksperiment med den egentlige $(1 + 1)$EA algoritme for at få det nøjagtige resultat.

6 Diskussion

Programmet blev bygget op som et framework hvor det er let at teste. Dette blev gjort ved at udnytte objekt-orienteret programmering, ved at fitness funktioner nedarver fra `FitnessFunction` og selektioner nedarver fra `Selection`. Nye fitness funktioner og selektioner kan så implementeres, uden at lave special tilfælde for nye kørsler af dem. Det ville også være muligt at implementere andre algoritmer ved at benytte samme metode som med fitness funktioner og selektioner.

Derefter blev der lavet en række eksperimenter, hvor hypoteser om de implemeterede fitness funktioner og selektioner blev påvist i alle tilfælde. Det må derfor antages, at de blev korrekt implementeret.

7 Konklusion

I projektet blev algoritmen $(\mu + 1)$EA implementeret, samt fire forskellige fitness funktioner og selektions mekanismer til løsning af optimeringsproblemer. Algoritmens køretider på forskellige inputs er blevet undersøgt gennem eksperimenter og resultaterne af disse eksperimenter viser, at de passer godt overens med de hypoteser, der blev opstillet. Fitness funktionen PreSuf, som aldrig før er blevet implementeret og undersøgt vha. en implemetering, viste sig at holde stik med hypoteser opstillet af Carsten Witt [2]. Den adfærd, der i litteraturen bevises for den oprindelige PreSuf funktion kan også bekræftes for den simplificerede version som er implementerede i programmet.

8 Videre arbejde med emnet

- Strukturer programmet bedre f.eks. ud fra Model-View-ViewModel eller Model-View-Control pattern.
- Konstruere Unit tests.
- Udvide programmet således, at det er muligt at udføre eksperimenter med algoritmerne $(\mu + \lambda)EA$ og $(1 + \lambda)EA$.
- Implementere og analysere andre former for populations baserede algoritmer.
- Implementere og analysere flere fitness funktions mekanismer.
- Implementere og analysere flere selektions mekanismer.
- Optimering af mutationshastighed
- Implementere og analysere flere mutations mekanismer.
- Implementere og analysere "mating" mekanismer.
- Undersøge PreSuf fitness funktionen ved en nøjagtig teoretisk implementering.
- Undersøge Fitness Proportional og Tournament selektionerne på PreSuf med store værdier for n og μ (f.eks. 1000).

A Implementering af teoretiske modeller

A.1 Fitness Funktioner

A.1.1 OneMax

```
public double f(boolean[] in) {
        int res = 0;
        // Remove if the same as goal
        for (int i = 0; i < in.length; i++) {
            if (in[i])
            {res++;}
        }
        return res;
    }
```

A.1.2 Leading Ones

```
public double f(boolean[] in)
    {
        double res = 0;
        for (int i = 0; i < in.length; i++) {
            if (in[i])
            {res ++;}
            else
            {break;}
        }
        return res;
    }
```

A.1.3 SPC

```
public double f(boolean[] in)
    {
        double res = in.length;
        double front = 0;
        double total = 0;
        boolean lead = true;
        for (int i = 0; i < in.length; i++)
        {
            if (in[i] && lead)
            {front++;}
            else
            {lead = false;}
            if(in[i])
            {total++;}
        }
        if (res == front)
        {return res * 2;}
        else if (front == total && front < res)
        {return res + 1;}
         else
        {return res - total;}
    }
```

A.1.4 PreSuf

```
public double f(boolean[] p)
    {
         boolean[] prefix = new boolean[m];
         boolean[] suffix = new boolean[l];
         for (int i = 0; i < n; i++)
         {
             if (i < m)
             {prefix[i] = p[i];}
             else
             {suffix[i - m] = p[i];}
         }
         int po = PO(prefix);
         int lso = LSO(suffix);
         if (po <= Math.ceil(2 * m / 3))
         {return po + n * n * lso;}
         else
         {return (n * n * l - n * Math.abs(po - b) + lso - l - 1);}
    }
```

A.2 Selektioner

A.2.1 Uniformly Random

```
public boolean[] s()
    {
         int rdm = (int) (Math.random() * p.my);
         return this.p.get(rdm);
    }
```

A.2.2 Fitness Proportional

```
public boolean[] s()
    {
         double[] vals = luckyNumberGenerator();
         double w = Math.random();
         if(w == 1.0)
         {return p.get(p.my-1);}
         for(int i = 0; i < p.my; i++)
         {
             if(w <= vals[i])
             {return p.get(i);}
         }
         return p.get((int)(w*p.my));
    }
```

A.2.3 Tournament

```
public boolean[] s()
    {
         ArrayList<boolean[]> tmp = new ArrayList<>();
         for (int i = 0; i < k; i++)
         {
             int r = (int) (Math.random() * p.my);
             tmp.add(p.get(r));
         }
         tmp = sort(tmp);
         boolean[] res = tmp.get(0);
         for (int j = 1; j < k; j++)
         {
             double cr = Math.random();
             double h = ch * Math.pow(ch, j-1);
             if (cr < h)
             {res = tmp.get(j);}
         }
         return res;
    }
```

A.2.4 Truncation

```
public boolean[] s()
    {
         int res = 0;
         double resVal = f.f(p.get(0));
         for (int i = 1; i < p.my; i++)
         {
             if (f.f(p.get(i)) > resVal)
             {
                 res = i;
                 resVal = f.f(p.get(res));
             }
         }
         return p.get(res);
    }
```

B Eksperiment Resultater

B.1 OneMax med Unifomly Random

$\mu \backslash n$	10	20	30	40	50	60	70	80	90	100
1	49	127	222	317	431	561	694	793	936	1.069
10	79	233	405	600	789	986	1.197	1.398	1.605	1.833
20	106	356	618	904	1.194	1.494	1.801	2.125	2.429	2.766
30	132	463	823	1.204	1.585	1.997	2.392	2.807	3.233	3.680
40	152	573	1.023	1.491	1.994	2.481	3.012	3.503	4.041	4.555
50	174	669	1.213	1.796	2.361	2.969	3.597	4.186	4.818	5.430
60	191	778	1.416	2.063	2.773	3.470	4.181	4.881	5.602	6.314
70	212	874	1.603	2.372	3.122	3.945	4.744	5.535	6.342	7.145
80	228	967	1.772	2.649	3.491	4.389	5.272	6.205	7.132	8.040
90	239	1.059	1.973	2.905	3.900	4.864	5.849	6.836	7.850	8.887
100	247	2.134	2.134	3.202	4.227	5.301	6.405	7.543	8.646	9.748

B.2 OneMax med Truncation

$\mu \backslash n$	10	20	30	40	50	60	70	80	90	100
1	46	125	229	329	447	562	681	811	935	1.057
10	38	118	211	311	425	532	657	774	923	1.043
20	33	110	211	312	413	542	650	772	910	1.036
30	32	111	201	309	413	528	646	770	893	1.046
40	32	109	199	308	413	533	657	780	916	1.022
50	32	107	204	301	415	528	654	798	895	1.043
60	26	108	200	301	408	528	654	774	892	1.022
70	28	102	202	298	401	529	635	760	880	1.026
80	27	104	201	302	417	531	647	758	894	1.025
90	25	107	196	301	414	536	635	781	877	1.018
100	25	105	198	298	405	511	650	770	875	1.030

B.3 OneMax med Fitness Proportional

$\mu \backslash n$	10	20	30	40	50	60	70	80	90	100
1	47	127	227	322	448	545	684	804	938	1.079
10	72	227	393	585	780	987	1.187	1.368	1.599	1.798
20	99	333	597	883	1.168	1.473	1.783	2.084	2.404	2.736
30	118	438	797	1.177	1.557	1.957	2.367	2.764	3.195	3.632
40	132	535	991	1.452	1.933	2.453	2.967	3.458	3.979	4.513
50	147	627	1.170	1.723	2.321	2.910	3.494	4.152	4.727	5.374
60	166	717	1.344	2.010	2.666	3.378	4.058	4.792	5.477	6.224
70	181	789	1.516	2.266	3.035	3.803	4.619	5.416	6.259	7.046
80	189	894	1.681	2.531	3.379	4.280	5.189	6.087	6.994	7.910
90	197	972	1.879	2.804	3.773	4.716	5.740	6.735	7.701	8.750
100	207	1.055	2.003	3.053	4.097	5.154	6.289	7.380	8.480	8.754

B.4 OneMax med Tournament

$\mu \backslash n$	10	20	30	40	50	60	70	80	90	100
1	47	128	216	330	441	562	665	813	940	1.059
10	70	213	371	536	711	898	1.089	1.275	1.474	1.690
20	93	307	538	788	1.047	1.322	1.585	1.870	2.134	2.442
30	118	395	709	1.036	1.384	1.717	2.062	2.432	2.819	3.166
40	134	484	881	1.286	1.697	2.127	2.554	3.007	3.462	3.893
50	124	474	855	1.256	1.685	2.111	2.543	2.966	3.402	3.849
60	132	536	990	1.453	1.906	2.394	2.908	3.385	3.893	4.416
70	122	509	919	1.366	1.809	2.282	2.744	3.228	3.696	4.207
80	135	540	1.031	1.518	1.993	2.505	3.040	3.565	4.078	4.607
90	117	527	956	1.418	1.873	2.358	2.849	3.355	3.840	4.352
100	122	550	1.031	1.528	2.026	2.549	3.086	3.623	4.147	4.689

B.5 Leading Ones med Unifomly Random

$\mu \backslash n$	10	20	30	40	50	60	70	80	90	100
1	82	343	776	1.352	2.156	3.105	4.219	5.510	6.952	8.535
10	118	478	1.033	1.747	2.696	3.748	5.004	6.385	8.041	9.869
20	162	674	1.410	2.380	3.505	4.816	6.339	7.958	9.762	11.810
30	495	844	1.773	2.967	4.300	5.943	7.602	9.687	11.750	14.032
40	227	1.040	2.151	3.516	5.130	7.006	9.003	11.264	13.682	16.224
50	253	1.185	2.521	4.081	5.942	7.961	10.349	12.859	15.678	18.429
60	266	1.349	2.862	4.652	6.763	9.107	11.703	14.455	17.527	20.821
70	284	1.487	3.149	5.206	7.567	10.190	13.048	16.104	19.508	23.000
80	307	1.647	3.444	5.758	8.338	11.135	14.375	17.789	21.306	25.185
90	314	1.796	3.815	6.243	9.037	12.243	15.639	19.341	23.258	27.528
100	336	1.910	4.105	6.778	9.813	13.206	17.001	20.929	25.075	29.810

B.6 Leading Ones med Truncation

$\mu \backslash n$	10	20	30	40	50	60	70	80	90	100
1	83	332	776	1.381	2.154	3.098	4.179	5.423	6.915	8.544
10	66	308	723	1.340	2.064	2.985	4.121	5.369	6.775	8.408
20	61	300	715	1.263	2.042	2.923	4.034	5.410	6.788	8.384
30	55	298	714	1.258	2.069	2.968	3.995	5.254	6.752	8.254
40	54	281	682	1.255	1.984	2.901	4.066	5.301	6.726	8.309
50	51	286	668	1.246	2.016	2.904	4.034	5.243	6.770	8.362
60	49	276	675	1.256	1.991	2.911	3.960	5.173	6.677	8.249
70	48	281	676	1.253	1.987	2.949	3.964	5.237	6.707	8.333
80	46	275	670	1.257	1.982	2.939	3.999	5.234	6.764	8.288
90	45	277	670	1.254	1.991	2.881	3.939	5.170	6.701	8.313
100	42	261	680	1.219	1.973	2.922	4.015	5.226	6.663	8.292

B.7 SPC med Unifomly Random

$\mu \backslash n$	10	20	30	40
1	942	9.254	32.914	79.547
10	2.723	39.819	163.438	419.708
20	3.566	69.178	272.571	750.794
30	3.687	89.064	371.810	1.054.837
40	4.020	107.357	491.512	1.342.384
50	4.002	124.156	594.632	1.618.160

B.8 PreSuf med Unifomly Random

$\mu \backslash n$	10	20	30	40	50	60	70	80	90	100	1000
1	605	699	818	877	935	952	976	963	980	985	1.000
10	244	379	473	499	562	577	632	625	715	655	1.000
20	73	205	311	378	403	402	468	363	405	363	996
30	17	110	245	300	345	339	359	261	272	250	984
40	4	48	164	253	300	285	286	179	183	138	949
50	1	37	111	208	262	282	285	161	162	101	876
60	0	14	72	153	224	229	252	155	129	87	766
70	0	5	47	126	217	229	254	133	135	69	638
80	0	3	37	101	188	200	222	109	133	66	509
90	0	2	21	85	167	159	235	109	106	59	369
100	0	3	11	59	149	153	178	115	98	53	282
1000	0	0	0	0	0	0	0	0	1	1	0*

B.9 PreSuf med Truncation

$\mu \backslash n$	10	20	30	40	50	60	70	80	90	100	1000
1	689	797	863	925	930	959	968	968	981	976	1.000
10	347	492	636	672	802	830	908	889	922	909	1.000
20	130	307	459	547	697	752	820	839	883	878	1.000
30	38	202	365	442	573	685	770	761	808	830	1.000
40	14	106	300	384	523	654	728	708	796	814	1.000
50	2	55	243	335	456	615	702	666	740	736	1.000
60	0	21	188	263	384	611	655	602	735	717	1.000
70	0	12	124	234	346	531	613	600	704	694	1.000
80	0	6	90	188	311	533	611	584	678	662	1.000
90	0	3	73	140	243	513	569	543	665	627	1.000
100	0	3	32	122	221	492	561	506	619	600	1.000
1000	0	0	0	0	0	5	23	83	268	170	227*

B.10 PreSuf med Fitness Proportional

$\mu \backslash n$	10	20	30	40	50	60	70	80	90	100
1	602	756	834	905	943	954	970	972	980	983
10	138	170	303	364	518	727	808	779	883	875
20	17	32	106	128	227	572	645	651	751	717
30	2	7	24	53	96	461	546	561	666	611
40	0	0	8	10	33	420	427	466	549	515
50	0	0	3	9	15	353	347	393	464	443
60	0	0	1	1	4	293	320	340	444	392
70	0	0	0	1	2	303	269	292	376	326
80	0	0	0	0	0	224	263	289	365	298
90	0	0	0	0	0	211	213	253	332	270
100	0	0	0	0	0	198	188	232	311	235

B.11 PreSuf med Tournament

$\mu \backslash n$	10	20	30	40	50	60	70	80	90	100
1	602	756	834	905	943	954	970	972	980	983
10	138	170	303	364	518	727	808	779	883	875
20	17	32	106	128	227	572	645	651	751	717
30	2	7	24	53	96	461	546	561	666	611
40	0	0	8	10	33	420	427	466	549	515
50	0	0	3	9	15	353	347	393	464	443
60	0	0	1	1	4	293	320	340	444	392
70	0	0	0	1	2	303	269	292	376	326
80	0	0	0	0	0	224	263	289	365	298
90	0	0	0	0	0	211	213	253	332	270
100	0	0	0	0	0	198	188	232	311	235

C Eksperiment Behandling

C.1 OneMax med Unifomly Random vs. teoretisk kørertid

$\mu \backslash n$	10	20	30	40	50	60	70	80	90	100
1	1,134	1,193	1,253	1,254	1,297	1,354	1,391	1,354	1,388	1,399
10	0,570	0,813	0,906	0,979	1,009	1,033	1,060	1,071	1,081	1,101
20	0,455	0,732	0,827	0,893	0,933	0,961	0,985	1,009	1,019	1,038
30	0,396	0,674	0,786	0,852	0,889	0,927	0,946	0,966	0,984	1,004
40	0,351	0,646	0,759	0,822	0,874	0,901	0,933	0,945	0,966	0,977
50	0,326	0,616	0,736	0,812	0,849	0,885	0,915	0,929	0,948	0,959
60	0,302	0,605	0,727	0,790	0,845	0,878	0,903	0,920	0,936	0,947
70	0,289	0,588	0,713	0,787	0,825	0,866	0,890	0,907	0,921	0,932
80	0,274	0,573	0,696	0,776	0,815	0,852	0,874	0,899	0,916	0,928
90	0,256	0,561	0,693	0,762	0,816	0,845	0,869	0,887	0,904	0,920
100	0,239	0,553	0,678	0,760	0,800	0,834	0,862	0,887	0,902	0,914

C.2 OneMax med Truncation vs.

$\mu \backslash n$	10	20	30	40	50	60	70	80	90	100
1	1,064	1,174	1,292	1,301	1,346	1,356	1,365	1,385	1,387	1,383
10	0,879	1,109	1,191	0,507	0,543	0,557	0,582	0,593	1,369	1,364
20	0,764	1,033	1,191	0,308	0,322	0,349	0,355	0,367	1,350	1,355
30	0,740	1,043	1,134	0,219	0,232	0,245	0,255	0,265	1,324	1,368
40	0,740	1,024	1,123	0,170	0,181	0,194	0,203	0,210	1,359	1,337
50	0,740	1,005	1,151	0,136	0,149	0,157	0,166	0,177	1,327	1,364
60	0,602	1,015	1,129	0,115	0,124	0,134	0,141	0,146	1,323	1,337
70	0,648	0,958	1,140	0,099	0,106	0,116	0,119	0,124	1,305	1,342
80	0,625	0,977	1,134	0,088	0,097	0,103	0,107	0,110	1,326	1,341
90	0,578	1,005	1,106	0,079	0,087	0,093	0,094	0,101	1,301	1,332
100	0,578	0,986	1,117	0,071	0,077	0,080	0,087	0,091	1,298	1,347

C.3 Leading Ones med Unifomly Random vs. teoretisk kørertid

$\mu \backslash n$	10	20	30	40	50	60	70	80	90	100
1	0,616	0,705	0,741	0,746	0,775	0,785	0,792	0,798	0,801	0,800
10	0,273	0,378	0,435	0,469	0,507	0,525	0,544	0,557	0,577	0,593
20	0,212	0,317	0,367	0,406	0,430	0,451	0,470	0,482	0,493	0,507
30	0,451	0,282	0,334	0,372	0,392	0,418	0,428	0,449	0,458	0,469
40	0,159	0,270	0,317	0,348	0,372	0,394	0,408	0,423	0,435	0,444
50	0,144	0,251	0,305	0,333	0,358	0,373	0,393	0,406	0,420	0,426
60	0,127	0,241	0,294	0,324	0,348	0,366	0,382	0,393	0,406	0,418
70	0,117	0,231	0,281	0,315	0,340	0,359	0,374	0,385	0,398	0,407
80	0,111	0,225	0,272	0,309	0,333	0,348	0,366	0,380	0,389	0,399
90	0,102	0,220	0,270	0,301	0,324	0,345	0,359	0,373	0,383	0,394
100	0,098	0,211	0,263	0,296	0,319	0,338	0,356	0,367	0,377	0,390

C.4 Leading Ones med Truncation vs.

$\mu \backslash n$	10	20	30	40	50	60	70	80	90	100
1	0,830	0,830	0,862	0,863	0,862	0,861	0,853	0,847	0,854	0,854
10	0,660	0,770	0,803	0,838	0,826	0,829	0,841	0,839	0,836	0,841
20	0,610	0,750	0,794	0,789	0,817	0,812	0,823	0,845	0,838	0,838
30	0,550	0,745	0,793	0,786	0,828	0,824	0,815	0,821	0,834	0,825
40	0,540	0,703	0,758	0,784	0,794	0,806	0,830	0,828	0,830	0,831
50	0,510	0,715	0,742	0,779	0,806	0,807	0,823	0,819	0,836	0,836
60	0,490	0,690	0,750	0,785	0,796	0,809	0,808	0,808	0,824	0,825
70	0,480	0,703	0,751	0,783	0,795	0,819	0,809	0,818	0,828	0,833
80	0,460	0,688	0,744	0,786	0,793	0,816	0,816	0,818	0,835	0,829
90	0,450	0,693	0,744	0,784	0,796	0,800	0,804	0,808	0,827	0,831
100	0,420	0,653	0,756	0,762	0,789	0,812	0,819	0,817	0,823	0,829

C.5 SPC vs. nedre teoretisk kørertid

$\mu \backslash n$	10	20	30	40
1	0	0	0	0
10	0,905	1,653	2,011	2,178
20	0,771	1,869	2,182	2,535
30	0,603	1,821	2,252	2,696
40	0,535	1,785	2,422	2,791
50	0,452	1,752	2,486	2,854

C.6 SPC vs. øvre teoretisk kørertid

$\mu \backslash n$	10	20	30	40
1	0,942	1,157	1,219	1,243
10	0,272	0,498	0,605	0,656
20	0,178	0,432	0,505	0,587
30	0,123	0,371	0,459	0,549
40	0,101	0,335	0,455	0,524
50	0,080	0,310	0,440	0,506

Litteratur

[1] Daniel W. Dyer. Evolutionary computation in java: A practical guide to the watchmaker framework, 2008-2010. http://watchmaker.uncommons.org/manual/index.html.

[2] Carsten Witt. Runtime analysis of the (μ+1)EA on simple pseudo-boolean functions. *Evolutionary Computation*, 14(1):65–86, 2006.