Sammendrag

IQ-tester har blitt brukt i en årrekke til å teste hvor intelligente mennesker er. Selv om det lenge har vært et mål å gjøre datamaskiner intelligente finnes det ingen lignende test for å sammenligne hvor langt utviklingen har kommet innen kunstig intelligens. Vi ønsker derfor å bruke IQ-tester laget for mennesker til å finne ut hvor godt en datamaskin løser slike oppgaver.

Hvor bra en datamaskin gjør det på en IQ-test er dermed et mål for hvor kunstig intelligent denne er. Ved å kjøre en lignende test igjen om noen år vil man kunne se hvordan utviklingen har vært innenfor de områdene som testen dekker.

IQ-tester inneholder svært varierte oppgaver innen områder som logikk, hukommelse og språkferdigheter, og ulike oppgavetyper krever ulike teknikker for å løses. Blant annet viser denne rapporten hvordan genetisk programmering kan brukes til å løse ulike oppgaver innenfor logikk. Denne teknikken er brukt til å løse to oppgavetyper, en der man skal finne det neste tallet i en tallrekke og en der man skal finne den neste figuren i en figurrekke. Kjøringene av implementasjonene viser strålende resultater, og bekrefter dermed at genetisk programmering er velegnet til å løse en del oppgaver som krever gode logiske evner.

For en rekke andre oppgaver inneholder rapporten en detaljert beskrivelse av hvordan disse kan løses. Blant annet er kunnskaper om naturlig språk og oppslag i ordbøker viktige elementer.

Forord

Denne oppgaven ble gitt av veileder Amund Tveit, og ble utført av undertegnede som masteroppgave i TDT4900, Datateknikk ved Institutt for Datateknikk og Informasjonsvitenskap, NTNU.

Jeg vil gjerne takke faglærer Magnus Lie Hetland, hovedveileder Amund Tveit og hjelpeveileder Rolv Inge Seehuus for kyndig veiledning og råd og tips i forbindelse med oppgaven. Uten dem hadde ikke rapporten vært det den er. Spesielt vil jeg takke hjelpeveileder for gode innspill i forbindelse med implementasjonen.

Trondheim, 8. juni 2005

Anders Lund Fredriksen

Innhold

Sammendrag i

Forord iii

1 Innledning 1

2 Motivasjon 3

3 IQ og IQ-tester 5
3.1 IQ . . . 5
3.1.1 IQ og biologiske sammenhenger . . . 7
3.1.2 Modeller for IQ . . . 7
3.1.3 Andre typer intelligens . . . 8
3.1.4 IQ og datamaskiner . . . 8
3.2 IQ-tester . . . 8
3.2.1 Hva er en IQ-test? Kritikk av IQ-tester . . . 9
3.2.2 Oppgavetyper . . . 10
3.2.3 Hjelpemidler . . . 12
3.2.4 Innlesning av IQ-tester. Kunnskapsrepresentasjon . . . 12

4 Genetisk programmering. Valg av programmeringsspråk og GP-rammeverk 17
4.1 Genetisk programmering (GP) . . . 17
4.1.1 Funksjoner og terminaler. Individer, populasjoner og generasjoner 18
4.1.2 Fitnessverdi og fitnessfunksjon . . . 20
4.1.3 Generering av nye generasjoner. Operatorer . . . 21
4.1.4 Termineringskriterium . . . 23
4.2 Valg av programmeringsspråk . . . 24
4.2.1 Java . . . 24
4.2.2 Alternativer til Java: C, C++ og Lisp . . . 24
4.3 Valg av GP-rammeverk . . . 25
4.3.1 Groovy Java Genetic Programming (JGProg) . . . 26
4.3.2 ECJ . . . 26

5 Tallrekker (implementasjon) 29
5.1 Forutsetninger og krav . . . 29
5.2 Rekketyper . . . 30
5.2.1 Lineære rekker . . . 31
5.2.2 To-alternerende rekker . . . 31
5.2.3 Flervariabelrekker . . . 33
5.2.4 "Jukserekker" . . . 33
5.2.5 Hvilket tall er det neste i en rekke? . . . 34
5.3 Beskrivelse av implementert system . . . 35
5.3.1 GP-funksjoner, -konstanter og -variabler . . . 35
5.3.2 GP-parametre . . . 36
5.3.3 Systemets virkemåte . . . 36
5.4 Resultater . . . 37
5.5 Mulige forbedringer . . . 37

6 Figurrekker (implementasjon) 39
6.1 Forutsetninger og krav . . . 40
6.2 Rekketyper . . . 40
6.2.1 Bevegelse . . . 41
6.2.2 Rotasjon . . . 41
6.2.3 Forandring . . . 42
6.3 Beskrivelse av implementert system . . . 43
6.3.1 Objektenes egenskaper . . . 44
6.3.2 Evaluering av egenskapene . . . 44
6.3.3 GP-funksjoner, -konstanter og -variabler . . . 47
6.3.4 GP-parametre . . . 48
6.3.5 Systemets virkemåte . . . 48
6.4 Resultater . . . 49
6.5 Mulige forbedringer . . . 50

7 Andre oppgavetyper 53
7.1 Språk . . . 53
7.1.1 Hvilket ord passer ikke inn? . . . 53
7.1.2 Sortere bokstaver til å danne ord . . . 54
7.1.3 Ordspråk . . . 55
7.1.4 Synonymer . . . 56
7.2 Hukommelse og konsentrasjon . . . 57
7.2.1 Hukommelse . . . 57
7.2.2 Konsentrasjon . . . 59
7.3 Logikk . . . 60
7.3.1 Figurrekke . . . 61
7.3.2 Tallrekke . . . 61
7.3.3 Bokstavrekke . . . 61

7.3.4 Matrise med figurer . . . 62
7.3.5 Ordpar . . . 62
7.4 Regning . . . 64
7.4.1 Tekstoppgaver . . . 64
7.5 Teknikk og romforståelse . . . 67
7.5.1 Teknikkoppgaver . . . 67
7.5.2 Speilbilder . . . 67
7.5.3 Utbrettede 3D-figurer . . . 68

8 Konklusjon **71**

A Kildekode, tallrekker **73**
A.1 Add.java . . . 73
A.2 Div.java . . . 74
A.3 I.java . . . 76
A.4 Mul.java . . . 77
A.5 Number.java . . . 79
A.6 NumberData.java . . . 88
A.7 NumberStatistics.java . . . 89
A.8 One.java . . . 90
A.9 Sub.java . . . 91
A.10 X1.java . . . 92
A.11 X2.java . . . 94
A.12 number.params . . . 95

B Kjøreeksempel, tallrekker **97**

C Resultater, tallrekker **99**

D Kildekode, figurrekker **103**
D.1 Add.java . . . 103
D.2 FigureData.java . . . 104
D.3 Figures.java . . . 105
D.4 Fourtyfive.java . . . 137
D.5 Mul.java . . . 138
D.6 One.java . . . 139
D.7 Sub.java . . . 141
D.8 Ten.java . . . 142
D.9 X.java . . . 143
D.10 figures.params . . . 144

E Kjøreeksempel, figurrekker **147**

F Resultater, figurrekker **151**

G Estimat av en datamaskins score på en IQ-test **157**

Figurer

3.1 Normalfordeling av IQ i intervallet mellom 50 og 150. 6
3.2 Tre figurer vi skal finne egenskapene til. 15

4.1 Flytskjema for genetisk programmering. 19
4.2 Program representert som et tre. 20
4.3 Mutert individ. 22
4.4 To individer før krysning. Krysningsnodene er grå. 23
4.5 De samme individene etter krysning. Krysningsnodene er grå. 23

6.1 Figurrekke som skal løses ved hjelp av implementasjonen. 39
6.2 Oppgave der *to* sirkler i en figur blir til én firkant i neste. 42
6.3 Oppgave der formen på objektet endrer seg mellom hver figur. 43
6.4 Oppgave med flere objekter per figur. 51

7.1 Oppgave med figurmatrise. 63
7.2 Oppgave med speilbilder. 68
7.3 Oppgave med utbrettede 3D-figurer. 69

F.1 Figurrekke brukt til testing, oppgave 1. 151
F.2 Figurrekke brukt til testing, oppgave 2. 152
F.3 Figurrekke brukt til testing, oppgave 3. 152
F.4 Figurrekke brukt til testing, oppgave 4. 152
F.5 Figurrekke brukt til testing, oppgave 5. 152

Tabeller

4.1 Treningseksempler for genetisk programmering. 18

6.1 Variabler i figurrekke-implementasjonen. 44
6.2 Sammenligning av parametre til ulike objekter. 51

7.1 Hukommelse: Matrise med bokstaver. 57
7.2 Konsentrasjonsoppgave. 60

C.1 Resultater, tallrekker, del 1. 100
C.2 Resultater, tallrekker, del 2. 101

F.1 Variabler i figurrekke-oppgave 1. 153
F.2 Variabler i figurrekke-oppgave 2. 153
F.3 Variabler i figurrekke-oppgave 3. 153
F.4 Variabler i figurrekke-oppgave 4. 154
F.5 Variabler i figurrekke-oppgave 5. 154
F.6 Resultater fra kjøring, oppgave 1. 154
F.7 Resultater fra kjøring, oppgave 2. 155
F.8 Resultater fra kjøring, oppgave 3. 155
F.9 Resultater fra kjøring, oppgave 4. 155
F.10 Resultater fra kjøring, oppgave 5. 155

G.1 Estimat av poeng fordelt på oppgavetyper. 158
G.2 Begrunnelse av oppgavetype-sannsynligheter, del 1. 159
G.3 Begrunnelse av oppgavetype-sannsynligheter, del 2. 160
G.4 Begrunnelse av oppgavetype-sannsynligheter, del 3. 161
G.5 Omregningstabell fra oppgavepoeng til IQ. 161

Kapittel 1

Innledning

I en årrekke har man, ved hjelp av såkalte IQ-tester, målt hvor intelligente mennesker er i forhold til hverandre. Personer som scorer godt på IQ-tester regnes for å være mer intelligente enn andre, og, som vi skal se senere i rapporten, regner man intelligente mennesker for å ha mer suksess i livet enn mindre intelligente mennesker.

I fem tiår har man forsøkt å lage datamaskiner med kunstig intelligens for å gjøre dem mer nyttige. I denne oppgaven vil i vi se på hvordan datamaskiner kan bruke teknikker fra kunstig intelligens til å løse IQ-tester laget for mennesker. Oppgavene i IQ-tester er varierte og egner seg derfor godt som et mål for å kunne si hvor god deler av den kunstige intelligensen til en datamaskin er.

Rapporten starter med å motivere for hvorfor vi bør la datamaskinen løse IQ-tester (kapittel 2) før vi i kapittel 3 gir et overblikk over hva som kjennetegner IQ og IQ-tester. Oppgaveteksten forutsetter at genetisk programmering skal brukes til å løse noen av oppgavene gitt i IQ-tester, og kapittel 4 forklarer derfor hva genetisk programmering er. Kapitlet inneholder også en oversikt over noen av programmeringsspråkene som egner seg for utvikling av systemer basert på genetisk programmering. Etter valg av programmeringsspråk følger en liten evaluering og valg av hvilket rammeverk for genetisk programmering som egner seg best for oppgavene vi skal løse.

I forbindelse med rapporten er det implementert to systemer for å løse ulike IQ-oppgaver. Det ene systemet finner hvilket tall som er det neste i en tallrekke (kapittel 5), mens det andre systemet finner hvilken figur som er den neste i en figurrekke (kapittel 6). I tillegg til beskrivelse av hvordan de to systemene er implementert inneholder de to ovennevnte kapitlene også en diskusjon av kjøreresultater og hvordan implementasjonene kan forbedres.

I kapittel 7 blir det beskrevet hvordan en rekke IQ-oppgaver ulike de som er implementert kan løses. Oppgavene, med beskrivelse av løsning, er primært hentet fra NRKs Test nasjonen-programmer fra 2003 [NRK03] og 2004 [NRK04c]. Disse testene er varierte og

har mange oppgaver, og representerer derfor godt hvordan en IQ-test kan være.

Til slutt, i kapittel 8, avslutter vi med å gi et estimat av hvor bra vi kan regne med at en datamaskin gjør det på IQ-tester dersom vi bruker metodene rapporten beskriver. Kapitlet avslutter med å se litt på veien videre, og hvilket arbeid som må gjøres for at datamaskinen skal gjøre det virkelig bra i IQ-tester.

Kapittel 2

Motivasjon

Å lage datamaskiner med god kunstig intelligens for å gjøre dem nyttigere har lenge vært et mål, men det finnes i dag ikke noe mål for hvor langt man har kommet i utviklingen av den. Derimot finnes det tester laget for å måle hvor intelligente mennesker er, nemlig IQ-tester. Selv om disse testene ikke er laget for at datamaskiner skal løse dem inneholder de likevel mange av problemstillingene som vi ønsker at kunstig intelligens skal yte godt på. Vi bruker derfor IQ-tester til å måle hvor god den kunstige intelligensen til en datamaskin er innenfor enkelte områder.

Ved å la datamaskinen løse en slik test kan man finne ut hvilke områder den gjør det bra på og hvilke områder den gjør det dårlig på. På den måten kan man finne ut innenfor hvilke domener man bør sette inn ressurser for å få datamaskinen til å yte bedre. Domenene inkluderer blant annet:

- Naturlig språk-forståelse.
- Bildebehandling.
- Logikk.
- Regning.
- Hukommelse.
- Kunnskapsrepresentasjon.
- Fysikk og teknikk.

Bedre ytelse innenfor de ovennevnte domenene vil ikke bare gi godt resultat i IQ-tester, men vil også komme vanlige folk til gode. For eksempel vil god naturlig språk-forståelse gjøre det lettere for en person å kommunisere med en datamaskin, og gode kunnskapsdatabaser innen fysikk og teknikk vil gjøre det enklere for folk å anvende kunnskaper de ikke selv besitter.

En pekepinn på hvor godt man ligger an får man dersom man utfører en IQ-test på en av dagens maskiner og gjør en lignende test igjen om noen år. Man vil da kunne se hvor god utviklingen har vært innenfor flere områder, blant annet de ovennevnte.

Kapittel 3

IQ og IQ-tester

I en årrekke har man målt hvor intelligente mennesker er i forhold til hverandre ved hjelp av IQ-tester [Man04]. Under følger et kort sammendrag av hva IQ er (kapittel 3.1), og hvordan IQ testes ved hjelp av IQ-tester (kapittel 3.2).

3.1 IQ

IQ, eller intellignskvotient, er et mål for intelligens. Det finnes mange definisjoner av intelligens, og det virker som mange er uenige om hva som er den beste forståelsen av ordet. Først skal vi se på noen objektive fakta omkring intelligens og IQ før vi skal se på hvordan intelligens er forsøkt definert.

IQ er i følge Ilstad [Ils02] definert som

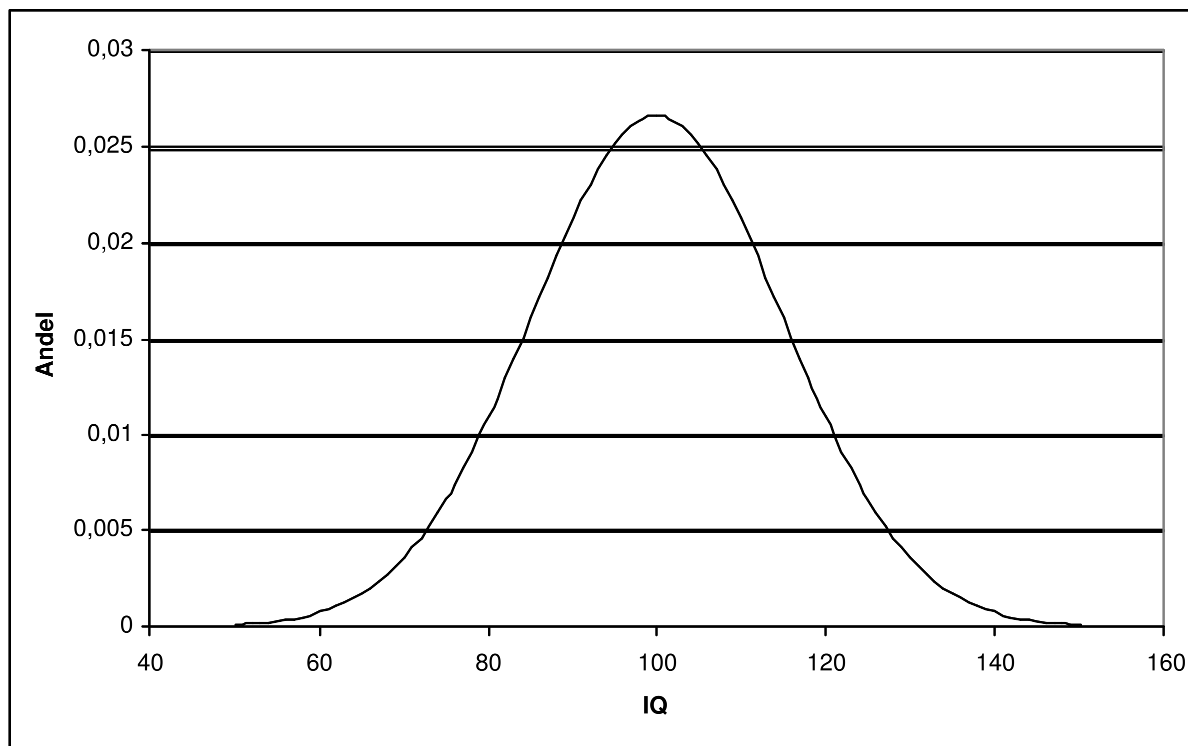
$$\frac{mental\ alder}{levealder} * 100.$$

Det gjør at dersom en person på X år er like intelligent som en gjennomsnittlig X-åring har han en IQ på 100. Dette målet på IQ er imidlertid ikke et godt mål fordi man ikke kan si at en person på 50 år med mental alder på 100 år har 200 i IQ, og definisjonen brukes derfor sjelden. Formelen over passer imidlertid godt for barn. En 10-åring som har mental alder som en 11-åring har 110 i IQ.

Gjennomsnittlig IQ er 100, og som mange andre ting i naturen er intelligensen normalfordelt, og standardavviket er 15 eller 16[1]. Så sant annet ikke er nevnt eksplisitt antar vi i denne rapporten at standardavviket er 15. Figur 3.1 viser skjematisk hvordan IQ er fordelt blant mennesker. Det gjør, som figuren viser, at 50 % av befolkningen har over

[1] Ulike IQ-tester har brukt ulike standardavvik (enten 15 eller 16). Denne forskjellen har selvsagt blitt sett på som beklagelig og gjør at man ikke uten videre kan sammenligne score fra to ulike IQ-tester.

100 i IQ, ca. 16 % har over 115, og ca. 2 % har over 130 hvis vi antar at standardavviket er 15.



Figur 3.1: Normalfordeling av IQ i intervallet mellom 50 og 150.

Selv om det er laget en rekke modeller for å prøve å beskrive IQ (se kapittel 3.1.2), er det mange som på en kort måte har prøvd å definere hva intelligens er. Manfeld lister i en artikkel følgende definisjoner [Man04]:

- Evnen til å resonnere blant annet ved å trekke konklusjoner basert på tidligere erfaringer.
- Et mål for suksess i livet.
- Det man måler med IQ-tester.
- Forstand, tenkeevne, skarpsindighet og fornuft.
- Summen av mennesket mentale evner.
- Hjernens evne til å samle inn, lagre og anvende informasjon.

At høyere IQ gir større suksess i livet høres politisk ukorrekt ut, men i følge en artikkel av Gottfredson stemmer dette [Got97]. Artikkelen hennes viser at personer med høy IQ

scorer bedre på en rekke faktorer[2] enn personer med lavere IQ, og dermed at personer med høy IQ har større suksess i livet enn personer med lav IQ. Se forøvrig Manfreds nevnte artikkel for videre diskusjon omkring de forskjellige definisjonene av intelligens [Man04].

3.1.1 IQ og biologiske sammenhenger

Både miljø og arv påvirker hvor høy IQ en person har. Dette bevises kanskje best ved at intelligensen til eneggede tvillinger er høyere korrelert dersom tvillingene er vokst opp sammen enn hvis de ble skilt fra hverandre tidlig i livet.[3]

Også mellom kjønn og raser er det forskjeller i IQ. I følge Manfred scorer asiater litt høyere enn hvite på IQ-tester, mens hvite scorer noe høyere enn sorte. Menn og kvinner scorer i gjennomsnitt like bra på IQ-tester, men scorer ikke like bra i de ulike kategoriene. For eksempel er menns matematiske evner bedre enn kvinners, mens kvinner er bedre på språklige områder. [Man04]

3.1.2 Modeller for IQ

Det finnes en lang rekker modeller som prøver å forklare hva IQ er. Den mest kjente av disse er g-faktor-modellen oppdaget av Spearman i 1904 [Kar04]. Modellen sier at et menneske har én generell faktor og mange spesifikke evner. Disse evnene er korrelert både med andre evner og med den generelle faktoren.

Hva denne generelle faktoren er har ikke Spearman noen formening om, men poenget med modellen er at en person med høye spesifikke evner innenfor et felt har stor sjanse for også å ha høye spesifikke evner innenfor andre felt også.[4] g-faktoren er i dag det samme som IQ, og måles som kjent ved hjelp av IQ-tester. Vi kan derfor si at en person som scorer godt på en IQ-test antageligvis gjør det bedre på et spesielt felt enn en som scorer dårlig på en IQ-test.

Det finnes en rekke modeller som kan sees på som supplementer til g-faktor-modellen. Disse inkluderer blant annet Sternbergs triarkiske modell, Cattells flytende/krystalliserte intelligens og Guttmanns Radex-modell. [Man04]

[2]Faktorerene inkluderer blant annet arbeidsløshet, fengselsopphold, skilsmisser, fattigdom og lav fødselsvekt hos egne barn.

[3]I følge Manfreds artikkel er korrelasjonen 0,86 for eneggede tvillinger som er vokst opp sammen, og 0,75 for eneggede tvillinger som ble skilt ved fødselen og har vokst opp i ulike miljøer [Man04].

[4]Vi kan sammenligne dette med skolefag. Korrelasjonen mellom karakterene i ulike fag for en og samme elev er nokså høy. Det er altså ikke vanlig at karakterene til en elev er jevnt fordelt over hele skalaen, men heller at de er nokså samlet, enten karakterene er gode, gjennomsnittlige eller dårlige.

3.1.3 Andre typer intelligens

I tillegg til den intelligensen som måles av IQ-tester finnes det en rekke andre, mer eller mindre allment aksepterte intelligenser. Den mest kjente av disse er nok emosjonell intelligens (se mer om emosjonell intelligens (EQ/EI) på eq.org [Fre05]). Gardner nevner i en av sine bøker at det antageligvis finnes 10-12 ulike typer intelligenser, men samme mann har forkastet at det finnes seksuell, spirituell og eksistensiell intelligens, intelligenser som har vært nevnt av flere tidligere [Gar99].

Så sant ikke annet er nevnt spesielt vil bruken av ordet intelligens i denne rapporten alltid bety den intelligensen som måles ved hjelp av IQ-tester. Se Gardners nevnte bok for videre diskusjon om andre typer intelligenser [Gar99].

3.1.4 IQ og datamaskiner

IQ er i utgangspunktet et mål for hvor intelligente mennesker er i forhold til hverandre. Dersom en person har en IQ høyere enn ett hundre er han mer intelligent enn en gjennomsnittlig person på hans alder. Fordi IQ-tester er laget for mennesker kan man ikke uten videre la for eksempel en datamaskin eller et dyr løse en slik test gitt til mennesker og på den måten få bestemt sin IQ.

Vi kan tenke oss at vi har et dyr som vi fra før vet er intelligent, men vi ønsker å måle hvor intelligent dette er i forhold til mennesker (vi ønsker altså å finne IQ-en til dyret). En IQ-test laget for mennesker vil fungere dårlig ettersom det er lite sannsynlig at dyret kan lese og forstå oppgaver på den måten. For at dyrene skal prestere så bra som de er i stand til må derfor oppgavene være tilpasset dem.

På samme måte vil det være riktigere å, når man skal måle IQ-en til en datamaskin, gi denne oppgaver som er tilpasset datamaskiner. For eksempel er datamaskiner dårligere til å forstå naturlig språk enn mennesker. Denne rapporten velger å ikke ta hensyn til at en datamaskin trenger helt andre oppgavetyper enn et menneske, og ser derfor heller på hvor godt en datamaskin løser en IQ-test laget for mennesker.

Som vi skal se senere i rapporten (kapittel 8) scorer en datamaskin mye ujevnere på en IQ-test enn et menneske. På oppgaver som omfatter for eksempel hukommelse vil en datamaskin være suveren.

3.2 IQ-tester

Mens det er over hundre år siden Spearman fant g-faktoren (kapittel 3.1.2) har IQ-tester i godt over femti år blitt benyttet for å måle intelligensen (inkludert denne g-faktoren) til mennesker. Testene brukes både i ansettelsesprosesser (selv om dette er mer populært

i USA enn i Norge [dSM94, side 7]), som underholdende tidsfordriv (for eksempel IQ-oppgaver gitt i diverse aviser og ukeblader) og for folk som enkelt og greit vil vite hvor intelligente de er.

IQ-testene består ofte av en rekke oppgavetyper. Dette kapitlet vil se litt på hva en IQ-test er og kritikk av noen tester (kapittel 3.2.1), og litt på hvilke oppgavetyper en IQ-test består av (kapittel 3.2.2). Kapitlet avsluttes med å ta for seg hvilke hjelpemidler datamaskinen vil ha nytte av når den skal løse IQ-tester (kapittel 3.2.3) før vi ser på hva som trengs for å gjøre en papirbasert IQ-test om til et format datamaskinen forstår (kapittel 3.2.4).

3.2.1 Hva er en IQ-test? Kritikk av IQ-tester

Det finnes en rekke ulike varianter av IQ-tester, noen mer seriøse enn andre. En av de mest seriøse tilgjengelige testene utføres av Mensa, en organisasjon for den delen av befolkningen med 2 % høyest IQ. Fordelen med Mensas test er at den er kulturfri, det vil si at man har mulighet til å klare testen uavhengig av nasjonalitet, religion, alder og så videre [Nor05].

Nettopp mangelen på kulturfrihet har gjort at NRK har blitt kritisert for IQ-testene gitt i programmet Test nasjonen. For eksempel vil en oppgave som består i å sammenligne betydningen av ulike ordspråk [NRK03, oppgave 11–14], [NRK04c, oppgave 12–17] favorisere ulike grupperinger. For eksempel vil en nordmann som er født og oppvokst i Norge ha bedre grunnlag for å besvare en slik oppgave enn en nordmann født og oppvokst i for eksempel Asia.

NRKs tester har også blitt kritisert fordi altfor stor del av testen måler hurtighet og oppmerksomhet [he05]. Det gjør at personer som vanligvis scorer høyt på IQ-tester, men som trenger litt tid til å tenke seg om scorer dårligere enn andre. Dette er en klar svakhet ved testen. I BBC program med samme navn (eng: Test The Nation) er enkelte av oppgavene enda tydligere på at det er hurtighet og oppmerksomhet som testes. Noen av oppgavene går her ut på å observere et bilde i noen få sekunder før man skal si hvilken feil bildet innehold [BBC05, oppgave 28–33].

Testene nevnt over har alle til felles at de kun måler IQ-en til en person innen et visst område. Det betyr at dersom man for eksempel scorer maksimalt av området testen gjelder for må man ta en annen test for å kunne bestemme IQ-en mer nøyaktig. En mye brukt test, både for å måle IQ-en til veldig intelligente mennesker og til å vurdere en persons evner i rettsaker, er WAIS (Wechsler Adult Intelligence Scale) og ulike varianter av denne [Wil05].

Som en liten kuriositet nevnte tidligere Mensa Norge-leder Ystenes i et nettmøte på Aftenposten følgende alternative IQ-test [YAd04]: Slipp en person med ukjent IQ inn i et rom med mange (for han) ukjente personer med kjent IQ og la han få prate med de

andre. Etter en stund spør man testpersonen hvem av de andre som var mest sosiale. Tar man denne personens IQ og trekker fra 5–10 poeng har man testpersonens IQ. Hvor godt en slik IQ-test egentlig fungerer skal være usagt.

3.2.2 Oppgavetyper

Det finnes en rekke ulike oppgavetyper brukt i IQ-tester. Denne rapporten vil spesielt konsentrere seg om oppgavetyper gitt i reelle IQ-tester med vekt på testene brukt i NRK-programmet Test nasjonen [NRK04b] i høsten 2003 [NRK03] og høsten 2004 [NRK04c]. Oppgavene i disse testene er varierte og spenner over et vidt spekter av oppgaver. I tillegg til disse testene er også en av BBCs tester fra samme program brukt [BBC05] og noen av IQ-test gitt i Séréville og Myers bok om hvordan man får suksess i IQ-tester [dSM94].

Under følger en oversikt over ulike oppgaver brukt i IQ-testene nevnt over. IQ-testene brukt i NRK-programmet Test nasjonen er delt i de fem hovedkategoriene språk, hukommelse og konsentrasjon, logikk, regning og teknikk og romforestilling, og den samme inndelingen er derfor brukt her.

De følgende avsnittene gir en kort beskrivelse av de ulike kategoriene og litt om hva slags oppgavetyper de inneholder. For en grundigere gjennomgang av oppgavetypene og hvordan disse kan løses ved hjelp av datamaskiner, se kapittel 7.

3.2.2.1 Språk

Språkkategorien er delt inn i fire underkategorier:

Hvilket ord passer ikke inn? Oppgaven inneholder fire ord, og består i å finne ut hvilket ord som har minst til felles med de andre.

Sortere bokstaver til å danne ord. Oppgaven inneholder fire lister med bokstaver i tilfeldig rekkefølge og en kategori (for eksempel frukt, sport eller lignende). Kun én av dem danner et ord i kategorien dersom man stokker listene.

Ordspråk. Oppgaven inneholder et ordspråk, og man skal fra en liste velge hvilket annet ordspråk som har samme betydning.

Synonymer. Hvilket ord betyr det samme som et gitt ord?

3.2.2.2 Hukommelse og konsentrasjon

Hovedkategorien er delt inn i to underkategorier:

Hukommelse. Først får man se et rutenett med tall, bokstaver, ord eller figurer i 30 sekunder. Oppgaven består i deretter å finne hvilket tall, bokstav, ord eller figur som ikke var i rutenettet.

Konsentrasjon. Oppgavene går ut på å se om ting er like, sette ord i alfabetisk rekkefølge og lignende. Med andre ord oppgaver som krever stor konsentrasjon.

3.2.2.3 Logikk

Logikkategorien er delt inn i fem underkategorier:

Figurrekke. Oppgaven inneholder en rekke med figurer. Hva er neste figur i rekken?

Tallrekke. Oppgaven inneholder en rekke med tall. Hva er neste tallet i rekken?

Bokstavrekke. Oppgaven inneholder en rekke med bokstaver. Hva er neste bokstav i rekken?

Matrise med figurer. Oppgaven inneholder en matrise med figurer der én figur mangler. Hvilken?

Ordpar. Eksempel: Papir står til saks som tre står til ...?

3.2.2.4 Regning

Regningkategorien består av én underkategori:

Tekstoppgaver. Et tekststykke som inneholder en regneoppgave.

3.2.2.5 Teknikk og romforståelse

Denne kategorien er delt inn i tre underkategorier:

Teknikkoppgaver. Veldig varierte oppgaver der blant annet kunnskaper om fysikk er nødvendig for å prestere bra.

Speilbilder. Oppgaven viser en rekke like, roterte figurer der én er speilvendt. Hvilken?

Utbrettede 3D-figurer. Oppgavene består i å sammenligne 3D-figurer med utbrettede 3D-figurer for å se hvilke figurer som er like.

3.2.3 Hjelpemidler

På vanlige IQ-tester er det ikke lov å bruke noen former for hjelpemidler annet enn blyant, viskelær og kladdepapir [Nor05]. Hvilke hjelpemidler som skal være tilgjengelige for en datamaskin er derimot et annet spørsmål. Datamaskinen må nødvendigvis ha tilgang til andre og flere hjelpemidler enn et menneske. For eksempel forstår ikke en datamaskin betyningen av noen ord uten en form for ordliste. Vi tillater derfor datamaskinen å bruke de hjelpemidlene (programvare og databaser) det er muligheter til å ha på én enkelt datamaskin. Kommunikasjon med omverdenen vil dermed ikke være lovlig.

Det er spesielt ett hjelpemiddel datamaskinen vil ha god bruk for, en database over ord kalt WordNet [Uni05]. WordNet er en samling bestående av de fleste engelske ord satt i et slags nettverk slik at man kan få oversikt over ulike relasjoner mellom ordene. Relasjonene omfatter blant annet synonymer, hyponymer[5] og hypernymer[6].

I tillegg til WordNet finnes det en annen nyttig og god database kalt KnowItAll [oW05]. KnowItAll er en stor samling av fakta samlet automatisk fra verdensveven. Man kan for eksempel finne ord som på en eller annen måte har tilknytning til andre ord. Hvis man søker etter det engelske ordet car får man opp en liste over ord relatert til ordet man søker etter. For car vil dette blant annet innbefatte ulike bilmerker. Å søke etter bilmerker i WordNet fungerer ikke fordi disse ikke er nevnt i ordboka.

Ettersom de ovennevnte databasene kun finnes for engelsk språk må oppgaver gitt på norsk oversettes før WordNet eller KnowItAll kan brukes. Som vi skal se senere (kapittel 7) gjelder dette stort sett bare enkeltord, og vi kan dermed bruke en vanlig norsk-engelsk ordbok slik som for eksempel Clue [ASA05].

3.2.4 Innlesning av IQ-tester. Kunnskapsrepresentasjon

For at en datamaskin skal kunne løse en IQ-test må IQ-testen være representert på en måte som gjør at datamaskinen forstår oppgavene. Stegene fra å ha IQ-testen på papir til ferdig digital form med oppgavene representert på en god måte er derfor en komplisert prosess. Prosessen kan deles i tre trinn, innskanning, tekst- og bildegjenkjenning (kapittel 3.2.4.1), naturlig språk-forståelse (kapittel 3.2.4.2) og kunnskapsrepresentasjon (kapittel 3.2.4.3).

Denne rapportens oppgave er kun å beskrive disse stegene på en overordnet måte. Når det senere skal implementeres systemer for å løse to oppgavetyper (kapittel 5 og 6)

[5] Hyponym er en mer spesiell betydning av et ord [JM00, side 601]. Bil er dermed et (av flere) hyponym til kjøretøy.

[6] Hypernym er en mer generell betydning av et ord [JM00, side 601]. Kjøretøy er dermed et (av flere) hypernym til bil, mens sportsgren er et (av flere) hypernym til rally.

forutsettes det at disse stegene allerede er løst.

3.2.4.1 Innskanning. Tekst- og bildegjenkjenning

Selve skanningen av en IQ-test er en enkel oppgave. Dagens skannere er såpass gode at det ikke er noe problem å få digitalisert det som står på et papirark.

Når innskanningen er gjort finnes det en rekke gode programvarealternativer for å gjenkjenne tekst og bilder. Disse programmene, som på engelsk kalles OCR (optical character recognition, se for eksempel about.coms liste over gode OCR-alternativer [Abo05]) er i stand til å oversette et innskannet dokument til et (redigerbart) tekstdokument. OCR-programmene skiller mellom tekst og bilder, og bevarer derfor eventuelle figurer som er en del av IQ-oppgaven.

3.2.4.2 Naturlig språk-forståelse

Etter at IQ-testen er skannet, gjort om til (redigerbar) tekst og figurene er skilt fra teksten må datamaskinen prøve å ”forstå” oppgavene. Naturlig språk-forståelse er en stor og komplisert prosess, men er likevel veldig viktig fordi både oppgaveteksten og ofte også selve oppgaveinnholdet består av tekst som skal tolkes. Naturlig språk-forståelse er derfor en av de viktigste delene når man skal løse IQ-tester.

Fordi naturlig språk-forståelse er så stort og omfattende vil det bare være mulig å omtale en liten del av det her. Utfordringene man står ovenfor når man skal lage et system for naturlig språk-forståelse inkluderer blant annet [JM00]:

- Lage en grammatikk slik at systemet vet hvilken rekkefølge ord kan komme i. Under følger en meget enkel og liten grammatikk, som blant annet godkjenner de to setningene ”Sjåføren kjører bilen” og ”Bilen kjører fort”:

  ```
  Setning -> SubstantivFrase VerbFrase
  SubstantivFrase -> Substantiv
  VerbFrase -> Verb SubstantivFrase
  VerbFrase -> Verb Adverb

  Substantiv -> bilen
  Substantiv -> sjåføren
  Verb -> kjører
  Adverb -> fort
  ```

  Grammatikken vil også godkjenne setninger som ikke nødvendigvis er riktig norsk, men som likevel er grammatisk riktige. ”Bilen kjører sjåføren” er eksempel på en slik setning.

- Tagging av ord slik at vi vet hva et ord betyr. Enkelte ord kan ha samme betydning. For eksempel kan baker både være verb (å bake) og substantiv (en baker). Setningen "Mor baker" er dermed kun riktig dersom vi tagger mor til substantiv og baker til verb. (Mor er entydig, og vil aldri være noe annet enn substantiv. Vi vet at en setning med to substantiver ikke er en fullstendig setning, og baker tagges derfor til å være verb.)
- Representere teksten på en måte som datamaskinen forstår ved hjelp av for eksempel predikatlogikk. Kapittel 7.4.1.1 viser hvordan dette kan gjøres i praksis.

Det er laget en rekke mer eller mindre gode systemer for naturlig språk-forståelse, men fellesnevneren for de fleste av disse er at de kun dekker et lite domene. Det finnes derfor, lite trolig, et ferdig system for naturlig språk-forståelse som kan brukes i forbindelse med IQ-tester.

Det finnes imidlertid en rekke rammeverk som kan brukes som grunnlag dersom man skal lage et slikt system for IQ-tester. Et av disse ble startet ved NTNU i 1991, og er et internet prosjekt ved IDI. Prosjektet kalles TUC (The Understanding Computer) og støtter både norsk og engelsk språk.

Det finnes for tiden tre applikasjoner som bruker TUC, BussTUC som er et spørresystem for ruteopplysning basert på naturlig språk (for mer om BussTUC, se artikler av henholdsvis Amble [Amb05] og Bratseth [Bra97], og for å teste systemet, se BussTUCs webside [NTN05a]), GeneTUC som er et system for informasjonsgjenfinning i molekylær biologi-artikler, og LexTUC, et system for spørsmål og svar i naturlig språk mot leksikon. [NTN05b]

3.2.4.3 Kunnskapsrepresentasjon

Når oppgaven er skannet, og teksten er delt fra bildene og "forstått" er det nødvendig å representere oppgaven på en fornuftig måte. Teksten har vi allerede representert (kapittel 3.2.4.2) og vi trenger derfor ikke gjøre noe mer med denne.

Bildene bør representeres på to ulike måter. Den ene er å lagre bildene som de er, piksel for piksel, uten å gjøre noen endringer. Dette er en enkel prosess, og vi trenger ikke behandle bildene noe mer for å lagre det på denne måten.

Den andre måten vi bør representere bildene på er å representere dem ved hjelp av egenskaper. Et bilde består ofte av flere ulike objekter, og det er derfor ønskelig å dele disse objektene fra hverandre og representere objektene ved hjelp av egenskaper. Egenskaper for objektene vi ønsker å ha med inkluderer blant annet objektets:

- Høyde og bredde.
- Plassering i figuren (objektets sentrum representert som X- og Y-koordinat).

- Farge.
- Form (for eksempel sirkel, trekant eller firkant).

Ut fra disse egenskapene kan man så utlede andre egenskaper. For eksempel kan det være ønskelig å finne ut rotasjonen i forhold til figurens sentrum. Dette kan blant annet være ønskelig dersom man vil sammenligne plasseringen til ulike objekter i ulike figurer.

Den første utfordringen vi står ovenfor når vi skal angi egenskapene til hvert objekt i en figur er å finne de ulike objektene og segmentere disse. Vi bruker de tre figurene i figur 3.2 for å illustrere.

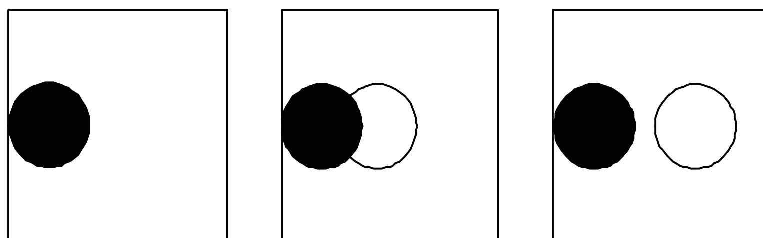
I figuren til venstre er det enkelt å finne alle de egenskapene som er listet over:

Høyde og bredde. Lag en firkant som dekker hele figuren. Gjør denne gradvis mindre slik at den til slutt er så liten som mulig, men slik at hele sirkelen fortsatt er inni firkanten. Sirkelens høyde og bredde er dermed firkantens høyde og bredde.

Plassering i figuren. Gjør som med høyde og bredde. Sirkelens sentrum er firkantens sentrum.

Form. Lag en liten trekant i sentrum av objektet og utvid denne gradvis. Trekanten må hele tiden være innenfor rammene av firkanten laget for å finne objektets høyde og bredde. Hvis man klarer å utvide trekanten så denne matcher objektets form er objektet en trekant. Passer det ikke må man prøve med trekanter av andre fasonger, og firkanter, sirkler og så videre.

Farge. Det finnes ulike måter å finne fargen i et objekt på. Dersom figuren er ensfarget kan man ta fargen i et enkelt punkt. En annen måte er å finne fargen til alle punktene i objektet og regne ut gjennomsnittsfargen. Dersom objektet består av ulike farger kan man segmentere objektet og finne fargene i de ulike områdene.



Figur 3.2: Tre figurer vi skal finne egenskapene til.

For figuren til høyre i figur 3.2 gjelder mye av det samme som for figuren til venstre. Den eneste forskjellen er at figuren består av to objekter. Man må derfor passe på at hvis en skal trekke en firkant rundt hvert objekt for å finne høyde og bredde må man passe på at firkanten kun dekker ett av objektene. Dette er ikke noe problem så lenge objektene er adskilte og fra hverandre.

Problemet oppstår når man skal finne egenskapene til den midterste figuren i figur 3.2. For de fleste mennesker er det, når man har sett figuren til venstre og høyre, ikke noe problem å se at figuren består av to sirkler der den ene er delvis dekket av den andre. Men hvordan skal datamaskinen vite om figuren består av ett objekt med to farger og noe uvanlig form eller to ensfargede sirkler?

En mulighet vil være å se på de to andre figurene i oppgaven og på den måten anta formen på et delvis skjult objekt. Ved å se på figuren til høyre (som inneholder to distinkte sirkler) kan vi anta at figuren i midten også inneholder to sirkler.

Thorsen har laget et meget godt program for å trekke ut egenskaper til objekter [Tho05]. Programmet vil løse mange av problemene omtalt i dette kapitlet, og er vel verdt å se nærmere på.

Kapittel 4

Genetisk programmering. Valg av programmeringsspråk og GP-rammeverk

Oppgaveteksten spør etter hvordan genetisk programming kan brukes til å løse enkelte IQ-oppgaver. Kapittel 4.1 inneholder derfor en overordnet beskrivelse av hva genetisk programmering er.

Videre diskuteres det hvilket programmeringsspråk implementeringen skal gjøres i (kapittel 4.2) og hvilket rammeverk som skal brukes for genetisk programmering (kapittel 4.3).

4.1 Genetisk programmering (GP)

Genetisk programmering er et stort og komplekst fagfelt, og det vil ikke være mulig å gi en grundig innføring i emnet i en rapport som denne. Det vil likevel bli gitt en kort oppsummering her, med en oversikt over hvordan genetisk programmering fungerer, og noen av de viktigste begrepene. For en dypere gjennomgang henvises det til diverse lærebøker [BNKF98], [Koz92].

Kort fortalt er genetisk programmering en automatisk måte å lage programmer på. Vi kan dele dele veien til løsning i fire steg [Fer05]:

1. Lag en initiell, tilfeldig populasjon med programmer bestående av funksjoner, variabler og konstanter (kapittel 4.1.1).
2. Kjør hvert program i populasjonen. Tilordne en fitnessverdi (kapittel 4.1.2) i forhold til hvor godt programmet løser problemet.

3. Lag en ny populasjon med programmer (kapittel 4.1.3). Denne populasjonen lages på grunnlag av de beste programmene fra forrige generasjon (kapittel 4.1.3.1), mutasjon (kapittel 4.1.3.2) og krysning (kapittel 4.1.3.3).

4. Fortsett å lage nye populasjoner (steg 2 og 3) helt til en har funnet et program som løser problemet bra nok (kapittel 4.1.4). Dette programmet er resultatet av den genetiske programmeringen.

For å få en bedre oversikt over hvordan stegene fungerer sammen er et flytskjema for genetisk programmering gjengitt i figur 4.1 [Fer05]. De påfølgende kapitlene vil forsøke å forklare stegene og flytskjemaet fra figuren.

For bedre å illustrere hvordan genetisk programming fungerer kan vi tenke oss et eksempel. Tabell 4.1 viser en oversikt over inndata (X og Y) og utdata. Vi ønsker å lage et program som, når det får inndata fra tabellen, skal produsere utdata som hører til inndataene. Inndata med tilhørende utdata kalles treningseksempler.

Tabell 4.1: Treningseksempler for genetisk programmering.

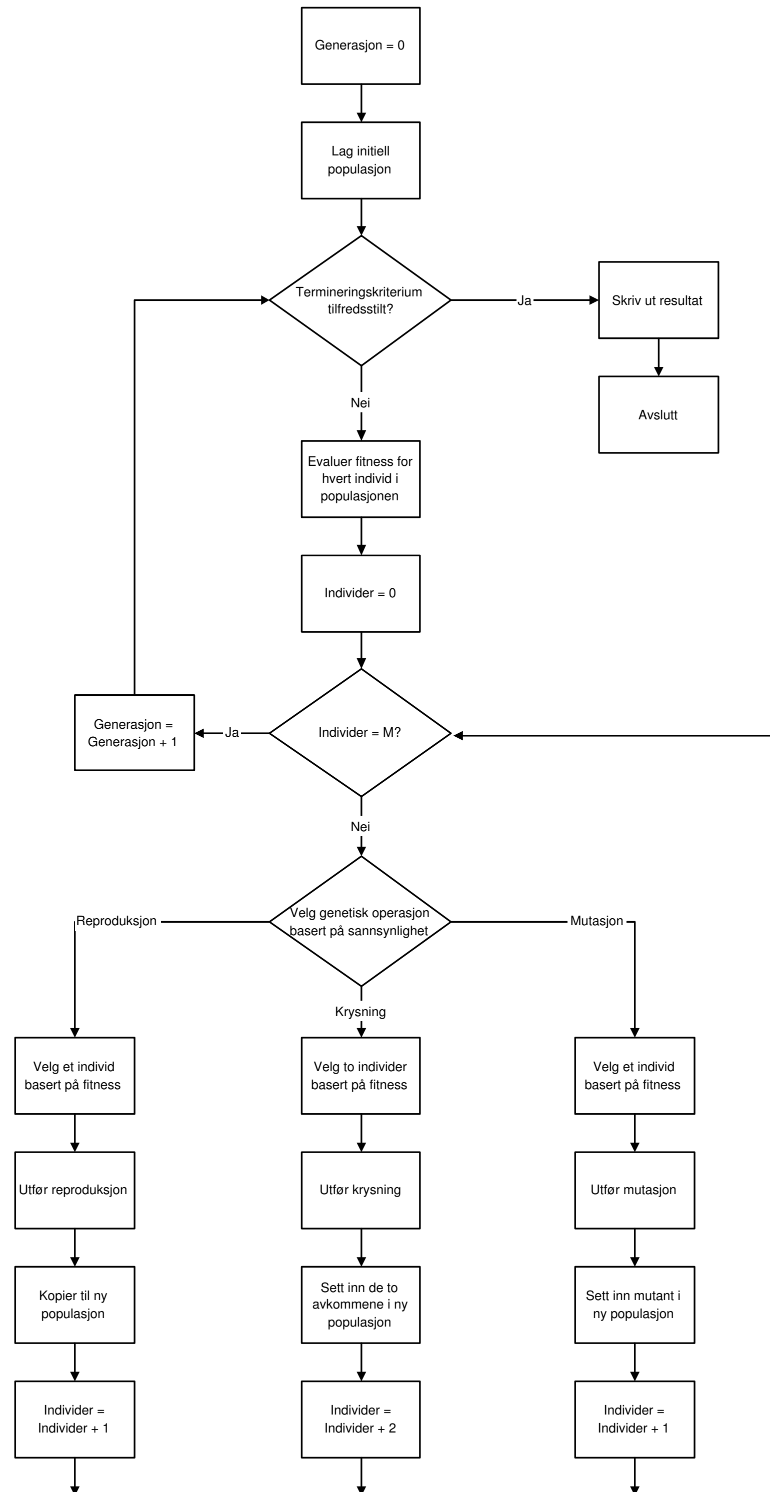
X	**Y**	**Utdata**
1	1	1
2	5	10
3	2	6

4.1.1 Funksjoner og terminaler. Individer, populasjoner og generasjoner

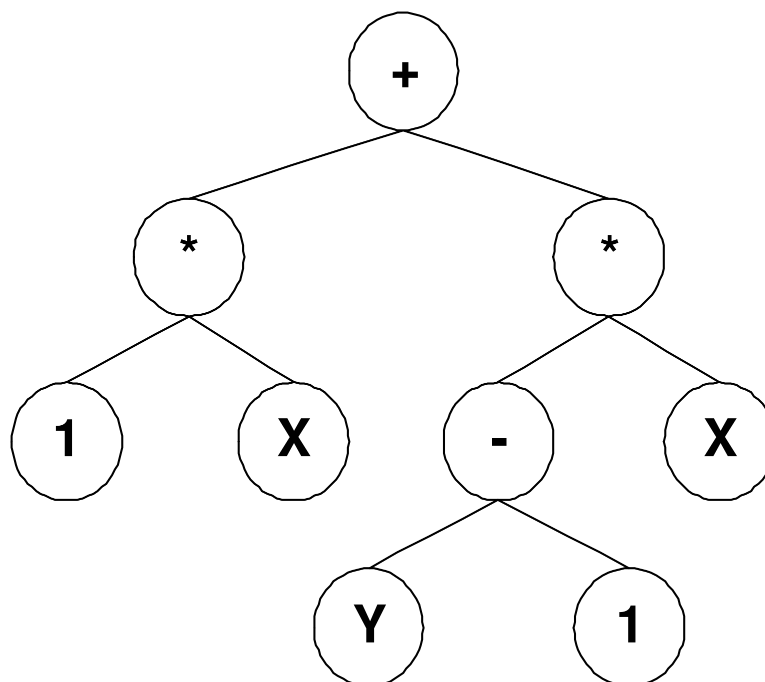
Vi kan tenke oss et program som et tre der hver node i treet enten er en funksjon eller en terminal (terminal er enten en variabel eller en konstant). Programmet i figur 4.2 er representert som et tre, og har tre tilgjengelige funksjoner (multiplikasjon, addisjon og subtraksjon), to tilgjengelige variabler (X og Y) og en tilgjengelig konstant (tallet 1).

Programmet kan også representeres som et regnestykke som $+ \ (* \ 1 \ X) \ (* \ (- \ Y \ 1) \ X)$ (polsk notasjon, parenteser kan dermed droppes), eller som $(1 * X) + ((Y - 1) * X)$ (algebraisk notasjon). Som vi ser er det ikke alle delene av programmet som nødvendigvis alltid er like konstruktive. $(* \ 1 \ X)$ kunne naturligvis vært forkortet til X, men GP har ikke som mål å lage så korte eller lesbare (for mennesker) programmer som mulig.

Et slikt program eller tre blir i genetisk programmering omtalt som et individ. Som første steget i genetisk programmering forteller kapittel 4.1 at en skal lage en initiell, tilfeldig populasjon. En populasjon består av en mengde individer (typisk i størrelsesorden rundt ett tusen). Når vi har laget så mange tilfeldige individer vi ønsker å ha i en populasjon har vi en generasjon.



Figur 4.1: Flytskjema for genetisk programmering.



Figur 4.2: Program representert som et tre.

4.1.2 Fitnessverdi og fitnessfunksjon

Steg to i genetisk programmering er å kjøre alle individene (programmene) i en generasjon på alle treningseksemplene og på det grunnlaget gi programmet en fitnessverdi. I eksemplet vårt har vi tre treningseksempler. For å regne ut fitnessverdien bruker vi følgende strategi:

1. Sett $n = 0$.
2. For hvert treningseksempel, gjør følgende:
   (a) Kjør programmet med de gitte inndata.
   (b) Regn ut den absolutte differensen mellom utdata fra programmet og gitte utdata, og legg denne til n (fitnessfunksjon).[1]
3. Fitnessverdien til individet normaliseres, blir dermed en verdi mellom 0 og 1, og

[1] Å finne den absolutte differensen mellom utdata fra programmet og gitte utdata er ikke nødvendigvis den beste måten å gjøre det på i dette tilfelle. En annen måte er å bare telle antall treff. Det er antageligvis en noe dårligere måte å gjøre det på siden antall treningseksempler er lite. (Dersom man teller antall treff må tallet inverteres på en måte ettersom lite tall er bedre. En mulighet er å trekke antall treff fra antall treningseksempler.)
En tredje mulighet hadde vært å kvadrere differensen mellom utdata fra programmet og gitte utdata. På den måten vil straffen være ekstra stor når avstanden differensen er stor.

er $fitness = \frac{1}{1+n}$.

Høyere fitnessverdi betyr at utdata fra programmet er likere gitte utdata enn lavere fitnessverdi.

Det finnes en rekke forskjellige måter å regne ut fitness på. Eksempelet over bruker en fitnessfunksjon basert på feil, og ser slik ut [BNKF98, kapittel 5.5]:

$$fitness = \sum_{i=1}^{n} |p_i - o_i|$$

der p_i er utdataene fra programmet og o_i er utdataene gitt i treningssettet.

En annen mye brukt fitnessfunksjon er basert på kvadrerte feil, og ser slik ut [BNKF98, kapittel 5.5]:

$$fitness = \sum_{i=1}^{n} (p_i - o_i)^2$$

der p_i er utdataene fra programmet og o_i er utdataene gitt i treningssettet.

En tredje fitnessfunksjon som brukes kalles skalerte kvadrerte feil, er omtalt i en artikkel av Keijzer og ser slik ut [Kei03]:

$$fitness = \sum_{i=1}^{n} (p_i - (a + b * o_i))^2$$

der

$$a = \overline{p_i} - b * \overline{o_i}$$

og

$$b = \frac{cov(p_i, o_i)}{var(o_i)}$$

og $\overline{x} = \frac{1}{n} * \sum_{i=1}^{n} x$.

Se Keijzers artikkel for hvordan man har kommet frem til denne fitnessfunksjonen og hvorfor den er bra.

4.1.3 Generering av nye generasjoner. Operatorer

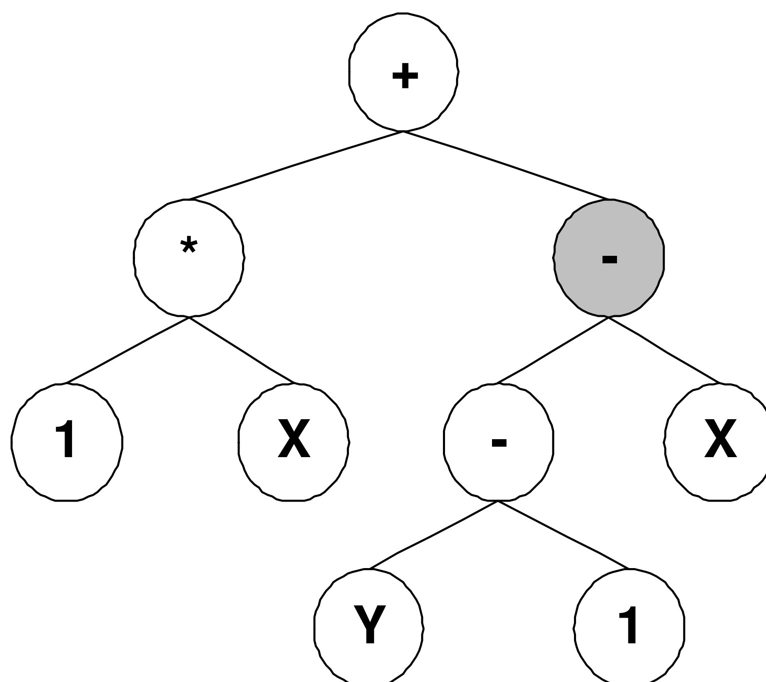
Steg tre i genetisk programmering er å lage nye populasjoner på grunnlag av tidligere populasjoner. Dette gjøres ved å la individene med høyest fitness danne grunnlaget for en ny generasjon. Oftest brukes i størrelsesorden 10–20 % av de beste individene fra en generasjon for å lage neste generasjon ved hjelp av reproduksjon (kapittel 4.1.3.1), mutasjon (kapittel 4.1.3.2) og krysning (kapittel 4.1.3.3).

4.1.3.1 Reproduksjon

Reproduksjon vil si å kopiere et individ fra en generasjon til neste uten å gjøre endringer. Dersom man reproduserer individet med høyest fitnessverdi sikrer man at den til en hver tid siste generasjon inneholder individet med høyest fitnessverdi.

4.1.3.2 Mutasjon

Mutasjon vil si å gjøre en liten endring i programmet. Endringen kan være å bytte ut en funksjon med en annen funksjon eller å bytte ut en terminal med en annen terminal. I figur 4.3 er en multiplikasjonsfunksjon byttet ut med en subtraksjonsfunksjon. Den aktuelle noden er markert med grått i figuren.



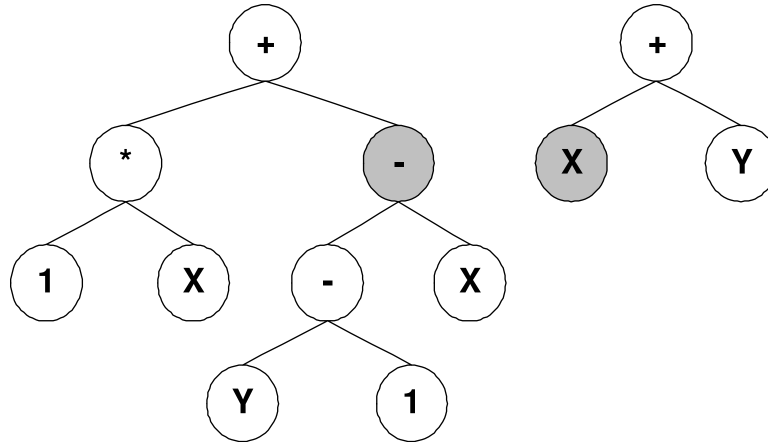
Figur 4.3: Mutert individ.

Dersom man muterer en funksjon med en annen funksjon er det viktig at funksjonene tar like mange parametere inn (det vil si har like mange barn i et tre).

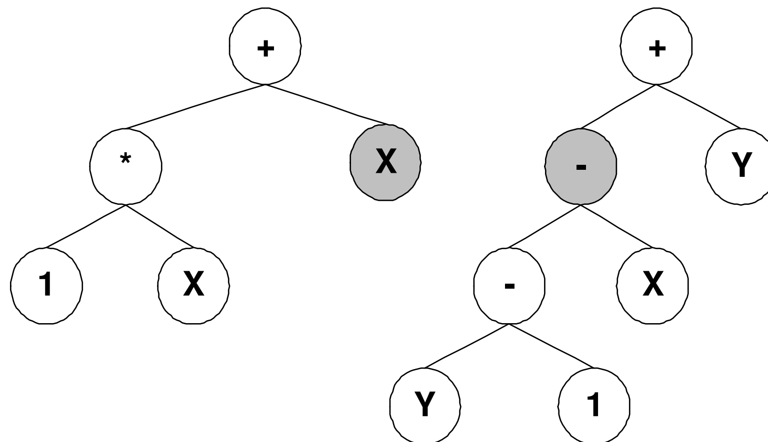
4.1.3.3 Krysning

Krysning vil si å bytte en node i et individtre med en annen node i et annet individtre. Figur 4.4 viser to individer som skal krysses. Nodene der trærne skal krysses er merket

med grått (disse bestemmes tilfeldig). Den grå noden med alle dens barn i det ene treet bytter plass med den grå noden med alle dens barn i det andre treet. Resultatet er vist i figur 4.5.



Figur 4.4: To individer før krysning. Krysningsnodene er grå.



Figur 4.5: De samme individene etter krysning. Krysningsnodene er grå.

4.1.4 Termineringskriterium

Det finnes to gode grunner til å avslutte den genetiske evolusjonen:

- Målet med evolusjonen er nådd. En har funnet et program som løser problemet enten perfekt eller godt nok.
- Vi har begrenset med tid og stopper derfor evolusjonen før den er ferdig.

I begge tilfeller vil individet med høyest fitnessverdi fra siste generasjon være løsningen av den genetiske programmeringen. Dersom $fitness = 1$ har vi funnet en løsning som passer perfekt til treningseksemplene.

4.2 Valg av programmeringsspråk

Det finnes en lang rekke programmeringsspråk å velge blant når man skal implementere et system ved hjelp av genetisk programmering [Wik05e], [Kin05]. Det viktigste kravet ved valg av programmeringsspråk er at det finnes et allerede eksisterende rammeverk for genetisk programmering fritt tilgjengelig, noe det gjør både for Java (omtalt i kapittel 4.2.1) og de alternativene til Java som er nevnt i kapittel 4.2.2.

4.2.1 Java

Java har siden lanseringen blitt et stadig mer populært programmeringsspråk. Tankegangen om at samme programvare skal kunne kjøres på en rekke ulike systemer har gjort Java til et av de meste brukte språkene, ikke bare til utvikling i forbindelse med Internett, men også i andre sammenhenger. Denne plattformuavhengigheten gjør at man kan utvikle og kompilere et program i Java på én plattform. Når programmet så er kompilert kan det brukes på et utall andre plattformer uten at man trenger å gjøre noen endringer [Wik05c].

Java har av mange blitt kritisert fordi programmer skrevet i språket kjører tregere enn programmer skrevet i for eksempel C og C++. Forskjellen er likevel ikke større enn at dette ikke vil bli oppfattet som noe problem. Et godt implementert GP-system vil kjøre raskere i Java enn et dårlig et vil gjøre i C/C++.

Disse fordelene, kombinert med at det, som nevnt i kapittel 4.3 allerede finnes mange ferdig implementerte rammeverk for genetisk programmering i Java gjør Java til et godt programmeringsspråk for vårt bruk. Java er derfor valgt som implementeringsspråk.

Java kommer stadig i nye versjoner. Versjon 5.0 er den nyeste, mens implementasjonen gjort i forbindelse med denne rapporten er gjort i versjonen før 5.0, nemlig versjon 1.4.2. For mer om Java, se Suns Java-websider [SUN05].

4.2.2 Alternativer til Java: C, C++ og Lisp

De mest aktuelle alternativene til Java er Lisp [Wik05d], C [Wik05b] og C++ [Wik05a]. Lisp er aktuelt fordi dette er et språk som brukes mye innenfor kunstig intelligens, og C og C++ er aktuelle fordi disse språkene allerede er meget utbredt og populære.

Lisp var det første funksjonelle programmeringsspråket, og ble opprinnelig laget til bruk innen kunstig intelligens [Set96, side 14]. På grunn av dette ble de første tre-basert GP-systemene laget i Lisp [BNKF98, side 295]. Lisps store fordel er språkets enkle syntaks. Jo enklere syntaksen er, jo lettere er det å sikre at krysning og mutasjon produserer syntaktisk riktige resultater. I Lisp må krysningsoperatoren i prinsippet kun sørge for at parentesene er balanserte.

Den største ulempen med Lisp er at språket er mye mindre utbrett enn de andre språkene nevnt i dette kapitlet. Skulle man laget et GP-system helt fra bunnen av hadde imidlertid Lisp vært et av de beste valgene. Ettersom vi velger å bruke et allerede ferdig implementert rammeverk til genetisk programmering vil ikke fordelene nevnt over komme til syne, og denne fordelen vil derfor ikke gjøre at Lisp velges som implementeringsspråk.

Forskjellene mellom C, C++ og Java er, sett i lys av hvilke systemer som skal implementeres i denne rapporten, små. Det finnes, som nevnt tidligere, også en lang rekke andre språk å velge mellom. Hvilket språk man velger av disse blir dermed en smaksak. Javas fordeler gjør at språket velges til fordel for C og C++.

4.3 Valg av GP-rammeverk

Det finnes en rekke rammeverk for genetisk programmering som allerede er implementert i Java, noe blant annet et søk på Google [Goo05] viser[2]. Mange av disse implementasjonene er listet nederst på ECJs webside [LPB+05].

GP-rammeverket som skal brukes i de videre implementasjonene trenger ikke være fullstendig eller komplett, eller støtte alle de mulighetene GP byr på. Viktigere er det at rammeverket er godt dokumentert, er raskt nok til at hastigheten ikke er så dårlig at denne virker som et problem, og at systemet har den funksjonaliteten *vi* trenger. Omfanget av denne rapporten er ikke stor nok til at vi kan teste mange rammeverk grundig mot hverandre. Taktikken som ble brukt for å finne et rammeverk ble derfor som følger:

- Test et rammeverk og se om det er godt nok dokumentert til at det er raskt å komme i gang med, og sjekk om det har den funksjonaliteten vi trenger.
- Så snart det viser seg at et rammeverk ikke støtter begge de ovennevnte kravene, test et nytt rammeverk.

Taktikken gjorde at det var nødvendig å teste to GP-rammeverk, Groovy Java Genetic Programming (se kapittel 4.3.1) og ECJ (se kapittel 4.3.2). Valget falt på sistnevnte.

[2]Et søk på websider/dokumenter som inneholder de tre ordene genetic, programming og java gir over 360 000 treff på Google.

4.3.1 Groovy Java Genetic Programming (JGProg)

Groovy Java Genetic Programming er, som navnet sier, en plattform for genetisk programming i Java [Bar00]. Systemet virket enkelt, noe som var årsaken til at nettopp dette rammeverket ble testet først.

Å komme i gang med helt enkle GP-programmer i JGProg er nokså lett fordi det følger med et knippe eksempelprogrammer. Problemet starter så snart man skal lage noe mer enn helt enkle programmer som ligner de vedlagte eksempelprogrammene. Årsaken til dette er at dokumentasjonen, med unntak av eksempelprogrammene og en sparsommelig Javadoc[3], er så og si ikke-eksisterende.

Det finnes blant annet ingen dokumentasjon på hvordan man lager sine egne funksjoner annet enn å se på kildekoden til allerede eksisterende implementerte funksjoner. Dette gjør det svært tidkrevende å lage egne funksjoner som skiller seg mye fra de allerede eksisterende.

Alt i alt er Groovy Java Genetic Programming et enkelt system som i beste fall egner seg til å leke med for bedre å forstå hva GP er, heller enn å implementere nyttige og mer komplekse GP-programmer. Ettersom utviklingen av JGProg ble lagt på is i første halvdel av 2000 er det neppe særlig sannsynlig at vi vil få se senere oppdateringer av systemet.

4.3.2 ECJ

ECJ er et system for evolusjonær beregning og genetisk programmering implementert i Java [LPB+05]. Ved første øyekast virker rammeverket godt dokumentert da det blant annet følger med fire tutorialer for å komme i gang med utvikling av GA[4] og senere GP. I tillegg til at systemet ved første øyekast tilsynelatende er godt dokumentert skryter gjengen bak ECJ på websiden sin av at systemet er veldig fleksibelt, lett å modifisere og raskt. Å teste rammeverket var derfor et naturlig valg.

ECJ består av et omfattende hierarki av paramterfiler. I parameterfilene settes ikke bare typiske GP-paramtere som populasjonsstørrelse og hvor mange generasjoner evolusjonen maksimalt skal foregå over, men også nesten alle Java-klassene og hvilke oppgaver de har settes der. Blant annet setter man i parameterfilene hvilke klasser som er programmets funksjoner (se kapittel 4.1.1). Bruken av paramterfiler sikrer dermed at man kan bestemme hvilke funksjoner programmet skal ha uten å måtte kompilere kildekoden på nytt.

Som nevnt mener utviklerne av ECJ at systemet er raskt. Blant annet skriver de at "ECJ

[3]Javadoc er Javas verktøy for å gjøre kode-kommentarer om til HTML-format.

[4]GA er en mye brukt forkortelse for genetiske algoritmer.

may make you reconsider notions about Java and slowness". Alle tester gjort i forbindelse med skrivingen av denne rapporten viser at ECJs skryt av hastighet er velbegrunnet.

Den eneste åpenbare ulempen med ECJ er at det, i forhold til hva vi egentlig trenger, er nokså stort og komplekst. Systemet virker som et komplett GP-system, og er derfor mer enn vi trenger. Men fordi systemet er så godt dokumentert som det er er dette ikke noe problem, snarere kanskje en fordel dersom man senere ønsker å utvide det man selv har laget.

Som kapitlet viser tilfredsstiller ECJ de kravene vi stilte til GP-rammeverket, og dette rammeverket er derfor valgt som grunnlag for senere implementasjon.

4.3.2.1 Viktige GP-parametre i ECJ

Som nevnt brukes parameter-filer i ECJ til å sette en rekke parametre, blant annet en del parametre som brukes i forbindelse med genetisk programmering. Disse parametrene har allerede fått tildelt en verdi i ECJ, og trenger i utgangspunktetet ikke endres. Som vi skal se senere i rapporten var det bare et lite fåtall av disse parametrene det var nødvendig å endre for å få implementasjonene i kapittel 5 og kapittel 6 til å fungere tilfredsstillende.

De viktigste GP-parametrene er:

Generasjoner. Forteller hvor mange generasjoner evolusjonen maksimalt skal kjøre. Dette tallet er av ECJ satt til 51, men tallet er økt i begge implementasjonene for å være mer sikker på at evolusjonen skal bli ferdig.

Antall subpopulasjoner. Som standard brukes kun én subpopulasjon per populasjon. Flere subpopulasjoner kan brukes dersom systemet skal evolvere flere ulike parametre samtidig.[5]

Individer i hver subpopulasjon. Antall indidvider i hver subpopulasjon er som standard satt til 1024.

Krysningssannsynlighet. 90 % av individene i en ny generasjon blir laget som følge av krysning.

Reproduksjonssannsynlighet 10 % av individene i en ny generasjon blir laget som følge av reproduksjon. Legg merke til at ingen av individene i en ny generasjon lages som følge av mutasjon.

Elitisme Som standard brukes ikke elitisme.

[5]Dette velger vi, av hastighetshensyn, likevel ikke å benytte oss av når flere parametre skal evolveres samtidig i figurrekke-systemet i kapittel 6.

For mer om valg av parametere i genetisk programmering, se boka Genetic Programming [BNKF98], og for mer om parametre i ECJ, se ECJs dokumentasjon om parameterfiler [ECJ05].

Kapittel 5

Tallrekker (implementasjon)

I en rekke IQ-tester består noen av oppgavene i å finne neste tallet i en tallrekke. Følgende oppgave, hentet fra Test nasjonen 2004 [NRK04c, oppgave 28] er eksempel på en slik tallrekke:

> 7, 11, 8, 12, 9, 13, ?
> a) 8
> b) 17
> c) 10
> d) 11

Oppgaven består i å erstatte spørsmålstegnet med et av svaralternativene a–d. Svaret her virker opplagt å være alternativ c fordi tallene i tallrekken henholdsvis øker med fire og minker med tre.[1]

Dette kapitlet inneholder en oversikt over et implementert system for å løse slike tallrekker. Kapittel 5.1 inneholder en liste med forutsetninger og krav til systemet, mens kapittel 5.2 inneholder en oversikt over ulike tallrekketyper. Kapittel 5.3 inneholder en beskrivelse av systemet som er implementert før kapittel 5.4 diskuterer kjøreresultater fra systemet. Til slutt, i kapittel 5.5 blir mulige forbedringer til det implementerte systemet diskutert.

5.1 Forutsetninger og krav

Som nevnt i kapittel 4.2 skal all implementasjon foregå i Java. Teknikken som skal brukes er genetisk programmering (se kapittel 4.1), og GP-rammeverket som skal benyttes

[1] Vi kan ikke umiddelbart si hvilket tall som er det neste i en tallrekke. Se kapittel 5.2.5 for videre diskusjon.

er ECJ (se kapittel 4.3).

Grunnlaget for implementasjonen er, som det står i kapittel 1, IQ-testen fra NRK-programmet Test nasjonen (2003 og 2004). Det betyr at programmet som implementeres skal kunne løse rekker av samme type og vanskelighetsgrad som oppgavene gitt i de nevnte testene. Med samme type menes at:

- tallrekkene skal inneholde seks tall, og oppgaven blir å finne det syvende.
- alle tallene i tallrekken er heltall.
- til hver oppgave blir det gitt fire svaralternativer der et og bare et av dem er det riktige.

Å definere hva som er samme vanskelighetsgrad som oppgavene i de ovennevnte IQ-testene er ikke lett, men en pekepinn kan være at alle oppgavene er av typer nevnt i kapittel 5.2.1, 5.2.2 eller 5.2.3. Oppgavene er heller ikke vanskeligere enn at de kan løses av mennesker uten noen form for hjelpemiddler (det skal for eksempel ikke være nødvendig å bruke kalkulator for å løse oppgavene).

5.2 Rekketyper

Det finnes en rekke ulike typer tallrekker. Når man vet hvilket system tallrekken bygger på, trenger man i de enkleste tallrekkene kun å vite ett tall i rekken for å kunne bestemme det neste. Et enkelt eksempel her er tallrekken 1, 2, 3, 4, 5, 6. Når man vet at et tall i rekken er det foregående inkrementert med én holder det å vite at det siste tallet er 6 for å finne løsningen.

De neste kapitlene vil ta for seg ulike rekketyper, hva som særpreger disse, og hvilke hensyn man må ta til ulikhetene når man skal lage et program som finner neste tallet i en tallrekke. Kapittel 5.2.1 tar for seg lineære rekker der et tall i rekken kun er avhengig av det foregående, kapittel 5.2.2 tar for seg alternerende rekker der et tall i rekken er avhengig av et av de foregående tallene i rekken (men ikke det som kommer umiddelbart før), kapittel 5.2.3 tar for seg rekker der et tall i rekken er avhengige av mer enn ett av de foregående tallene, mens kapittel 5.2.4 tar for seg rekker som det trengs en spesiell kunnskap for å løse.

Til slutt, i kapittel 5.2.5, vil si se på hvilke kriterier det stilles til det neste tallet i en tallrekke (det vil si hvordan vi kan vite hvilket nestetall som er det riktige).

5.2.1 Lineære rekker

Lineære rekker (slik uttrykket blir brukt i denne rapporten) er rekker som kan skrives ved hjelp av en av to av de følgende funksjonene:

- $f(x_i) = x_{i+1}$
- $f(x_i, i) = x_{i+1}$

Hvert tall i tallrekken er med andre ord kun avhengig av tallet rett forut for seg selv (og eventuelt plassering i rekken) for å tilfredsstille kravet som stilles til lineære rekker. Noen eksempler på rekker som tilfredsstiller $f(x_i) = x_{i+1}$ er:

- 1, 2, 3, 4, 5, 6 (+1)
- 1, 2, 4, 8, 16, 32 (*2)
- 5, 9, 17, 33, 65, 129 (*2-1)

Som vi ser tilfredsstiller disse rekkene også $f(x_i, i) = x_{i+1}$. Det finnes en del rekker som tilfredsstiller $f(x_i, i) = x_{i+1}$ men ikke $f(x_i) = x_{i+1}$. Eksempler på slike rekker er:

- 5, 7, 11, 17, 25, 35 (+2, +4, +6, osv.)
- 1, 1, 2, 6, 24, 120 (*1, *2, *3, osv.)
- 3, 5, 8, 13, 22, 39 (*2-1, *2-2, *2-3, osv.)

Ettersom alle lineære rekker tilfredsstiller $f(x_i, i) = x_{i+1}$ trenger vi ikke ta hensyn til om en rekke tilfredsstiller $f(x_i) = x_{i+1}$ eller ikke.

5.2.1.1 Hvordan løse lineære rekker?

Dersom vi vet at tallrekken er lineær, og tallrekken ligger innenfor den vanskelighetsgraden som er vanlig for tallrekker i IQ-tester er dette en nokså enkel oppgave. For å finne ut hvilket tall som er det neste i en lineær rekke må vi finne funksjonen $f(x_i, i) = x_{i+1}$ som passer for $0 < i < n$, der n er antall tall i tallrekken.

Den store utfordringen ligger i å finne ut om en rekke er lineær eller ikke. Hvordan dette kan løses kommer vi tilbake til i beskrivelsen av hvordan systemet er implementert (kapittel 5.3).

5.2.2 To-alternerende rekker

To-alternerende rekker (slik uttrykket blir brukt i denne rapporten) er rekker som kan skrives ved hjelp av et av to av de følgende funksjonsparene:

- $f_1(x_i) = x_{i+2}$ for $i = 2k + 1$
  $f_2(x_i) = x_{i+2}$ for $i = 2k$
- $f_1(x_i, i) = x_{i+2}$ for $i = 2k + 1$
  $f_2(x_i, i) = x_{i+2}$ for $i = 2k$

Som formlene over viser er et tall i tallrekken kun avhengig av tallet som er to plasser tidligere i rekken (og eventuelt plassering i rekken) for å tilfredsstille kravet som stilles til alternerende rekker[2]. Noen eksempler på rekker som tilfredsstiller $f_1(x_i) = x_{i+2}$ for $i = 2k + 1$ og $f_2(x_i) = x_{i+2}$ for $i = 2k$ er:

- 5, 8, 6, 9, 7, 10 (+3, -2)
- 9, 14, 28, 33, 66, 71 (+5, *2)
- 4, 16, 8, 32, 16, 64 (*4, /2)

Som vi ser tilfredsstiller disse rekkene også $f_1(x_i, i) = x_{i+2}$ for $i = 2k + 1$ og $f_2(x_i, i) = x_{i+2}$ for $i = 2k$. Det finnes en del rekker som tilfredsstiller $f_1(x_i, i) = x_{i+2}$ for $i = 2k + 1$ og $f_2(x_i, i) = x_{i+2}$ for $i = 2k$ men ikke $f_1(x_i) = x_{i+2}$ for $i = 2k + 1$ og $f_2(x_i) = x_{i+2}$ for $i = 2k$. Eksempler på slike rekker er:

- 1, 2, 4, 6, 12, 15, 30 (+1, *2, +2, *2, +3, *2, osv.)
- 14, 6, 18, 11, 33, 27 (-8, *3, -7, *3, -6, *3, osv.)
- 57, 66, 58, 65, 59, 64 (+9, -8, +7, -6, +5, -4, osv.)

Ettersom alle alternerende rekker tilfredsstiller $f_1(x_i, i) = x_{i+1}$ for $i = 2k+1$ og $f_2(x_i, i) = x_{i+1}$ for $i = 2k$ trenger vi ikke ta hensyn til om en rekke tilfredsstiller $f_1(x_i) = x_{i+1}$ for $i = 2k + 1$ og $f_2(x_i) = x_{i+1}$ for $i = 2k$ eller ikke.

5.2.2.1 Hvordan løse to-alternerende rekker?

Fremgangsmåten for å løse to-alternerende rekker har veldig mye til felles med hvordan man løse lineære rekker (kapittel 5.2.1.1). Fordi et tall i en to-alternerende rekke kun er avhengig av tallet to plasser tidligere i rekken gjør det at antall treningseksempler blir maksimalt halvparten av antall treningseksempler i en lineær rekke. Dersom vi har en rekke med seks tall og skal finne det syvende har vi dermed kun to treningseksempler, nemlig $f(x_1, 1) = x_3$ og $f(x_3, 3) = x_5$. Oppgaven vil da være å finne $f(x_5, 5) = x_7$.

At antall treningseksempler er lite gjør alternerende rekker vanskeligere å løse enn lineære rekker. Men på samme måte som for lineære rekker er den største utfordringen å finne ut om en rekke er alternerende eller ikke.

[2]Vi kan også tenke oss rekker som er avhengige av tallet som er for eksempel tre eller fire plasser tidligere i rekken. Vi velger imidlertid å se bort i fra disse i denne rapporten da rekkene i oppgavene er så korte at slike rekker vil få for få treningseksempler.

5.2.3 Flervariabelrekker

Det finnes rekker der et tall i rekken er en funksjon av flere enn ett av de foregående tallene. Disse rekkene vil i denne rapporten bli kalt flervariabelrekker, og to eksempler på slike rekker er:

- 2, 3, 5, 8, 13, 21 ($x_{i+2} = x_i + x_{i+1}$)
- 1, 2, 2, 4, 8, 32 ($x_{i+2} = x_i * x_{i+1}$)
- 15, 11, 4, 7, -3, 10 ($x_{i+2} = x_i - x_{i+1}$)

Problemet med slike rekker (i motsetning til rekkene nevnt i tidligere kapitteler) er at vi ikke kan vite hvor mange variable funksjonen har, og vi kan derfor ikke skrive opp en enkelt funksjon som tallrekkene tilfredsstiller. Det gjør at vi, dersom vi vet at en rekke er en flervariabelrekke, må gjette på hvor mange av de foregående tallene et tall er avhengige av.

Hvis vi antar at et tall i tallrekken er avhengig av de to foregående tallene og plasseringen i tallrekken, vil et tall i tallrekken kunne beskrives med følgende funksjon: $f(x_i, x_{i+1}, i) = x_{i+2}$

Denne kan skrives mer generelt, slik at den tar hensyn til at vi ikke vet antall variaber: $f(x_i, x_{i+1}, ..., x_i, i) = x_{i+n}$

5.2.3.1 Hvordan løse flervariabelrekker?

For å løse rekkene vi i dette kapitlet omtaler som flervariabelrekker brukes mye av den samme tankegangen som når lineære rekker skal løses (kapittel 5.2.1.1). Den største forskjellen er at vi ikke kan vite hvor mange av de foregående tallene et tall er avhengig av. Som nevnt i kapittel 5.1 består oppgavene alltid av 6 tall. Det er derfor lite trolig at et tall i en tallrekke er avhengig av mer enn to variabler.

På samme måte som for lineære rekker og alternerende rekker er ikke den største utfordringen å finne ut hvilken funksjon som tilfredsstiller en rekke vi vet er en flervariabelrekke, men heller å finne ut om en rekke faktisk er en flervariabelrekke eller ikke.

5.2.4 "Jukserekker"

Det finnes rekker som ikke passer inn under noen av de foregående overskriftene. Dette er rekker som man trenger spesiell kunnskap for å løse, og det er derfor lite ønskelig å bruke disse i IQ-tester. Å løse slike rekker er for en datamaskin meget vanskelig fordi

den da må ha tilgang på enorme mengder informasjon. La oss først se på et par eksempler før vi i kapittel 5.2.4.1 kommer med et forslag til hvordan sannsynligheten for å gi ut riktig svar blir noe større enn et tilfeldig gjett:

- 31, 28, 31, 30, 31, 30 (antall dager i måneden)
- 1, 5, 9, 15, 21, 25 (vokalplassering i alfabetet)
- 2, 2, 3, 4, 3, 4 (antall bokstaver i tallordene: en, to, tre, fire, osv.)

5.2.4.1 Hvordan løse ”jukserekker”?

Dersom vi vet at en rekke ikke kan løses ved hjelp av noen av de foregående metodene kan vi anta at rekken er en ”jukserekke” og vi kan da prøve å finne en løsning ved hjelp av tilnærming. Avsnittene under skiller mellom rekker som er voksende/synkende, og rekker som ikke er det under valg av løsningsmetode.

I rekker som er (strengt) voksende eller (strengt) synkende er det sannsynlig at også det neste tallet i tallrekken vil være henholdsvis større eller mindre enn det foregående. Vi kan derfor prøve å lage den lineære funksjonen som passer best til tallrekken og på den måten komme med et kvalifisert gjett. Dersom oppgaven inneholder svaralternativer velger vi det svaralternativet som ligger nærmest det kvalifiserte gjettet.

For rekker der tallene tilsynelatende øker og minker tilfeldig fra et tall i rekken til det neste er det noe vanskeligere å finne en tilnærming. Én mulighet er å la svaret være typetallet, det vil si det tallet som forekommer oftest i rekken. En annen mulighet kan være å ta gjennomsnittet av tallene i rekken og la det være svaret man gjetter på. En tredje, og mer kompleks måte å gjøre det på er å lage en funksjon som tilnærmer rekken, men heller ikke dette er en veldig god måte å løse problemet på.

Som vi ser er det vanskelig å finne noen god måte å løse rekkene vi i dette kapittelet har kalt ”jukserekker”. Fordi rekkene krever spesiell kunnskap for å løses og dermed sjeldent forekommer i IQ-tester er det derfor mindre farlig om vi har problemer med å løse disse rekketypene enn rekketypene nevnt i de foregående kapittelene. ”Jukserekker” er derfor ikke tatt hensyn til under implementasjonen.

5.2.5 Hvilket tall er det neste i en rekke?

Det er ikke umiddelbart gitt hvilket tall som *må* være det neste i en tallrekke. La oss se på et eksempel: Vi skal finne det neste tallet i følgende tallrekke: 1, 2, 3, 4. Et naturlig svar vil være 5, men hva om den opprinnelige tallrekken er som følger: 1, 2, 3, 4, 3, 2, 1, 2, 3, 4, 3, 2, 1. Eller for den saks skyld som følger: 1, 2, 3, 4, 1, 2, 3, 4, 1, 2, 3, 4. Vi kan med andre ord ikke si sikkert hvilket tall som er det neste, bare anta.

På AT&Ts webside om heltallssekvenser [AT&05] finnes en liste over mulige fortsettelser av en rekke. I IQ-tester er det vanlig å bruke prinsippet til Occam's Razor [Nil98, side 52]. For tallrekker vil det si å bruke den formelen som beskriver tallrekken på kortest mulig måte. Et tallrekken over (1, 2, 3, 4) kan beskrives ved hjelp av formelen $x_{i+1} = x_i + 1$, og 5 er derfor det femte tallet i tallrekken.

5.3 Beskrivelse av implementert system

I dette kapitlet finnes en oversiktlig beskrivelse av systemet implementert for å finne det neste tallet i en tallrekke. Systemet er, som nevnt i kapittel 4, implementert ved hjelp av genetisk programmering i Java. Kildekoden til systemet finnes på den vedlagte CD-en og i vedlegg A, mens et kjøreeksempel finnes i vedlegg B.

5.3.1 GP-funksjoner, -konstanter og -variabler

Systemet består av fire funksjoner, én konstant og tre variabler. Generelt om funksjoner, konstanter og variabler finnes i kapittel 4.1.1.

Systemets fire funksjoner er:

Add.java Legger sammen to tall.

Mul.java Multipliserer to tall.

Sub.java Trekker et tall fra et annet.

Div.java Dividerer to tall.

Systemets ene konstant er:

One.java Heltallet 1.

Systemets tre variabler er:

X1.java X1 er et tall i rekken.

X2.java X2 er tallet etter X1.

I.java I er plasseringen til et tall i rekken.

Årsaken til at vi trenger tre variabler er for å løse tallrekke-oppgaver lik de som er beskrevet i kapittel 5.2.2. Selv om tre variabler er tilgjengelige brukes til en hver tid kun de variablene systemet trenger under evolusjonen.

5.3.2 GP-parametre

Alle GP-parametrene brukt i implementasjonen finnes i såkalte paramter-filer. Med to unntak har alle GP-parametrene brukt i forbindelse med implementasjonen de verdiene som er standard for ECJ (de viktigste av disse diskutert i kapittel 4.3.2.1). De to parametrene som ikke har standardverdier er:

Elitisme De 3 beste individene fra hver generasjon går videre til neste. Dette sikrer blant annet at uansett når evolusjonen stopper vil de tre beste individene totalt sett finnes i den siste generasjonen.

Generasjoner. Antall generasjoner er satt til 100. Dersom systemet ikke har funnet et perfekt individ etter 100 generasjoner avsluttes evolusjonen og det beste individet velges.

Generatoren som genererer tilfeldige tall blir sådd med tid slik at hver kjøring blir tildelt tall ulik de forrige kjøringene.

5.3.3 Systemets virkemåte

Dette kapitlet inneholder en beskrivelse av den delen av systemet som er et tillegg til GP-rammeverket (ECJ) fungerer, og altså ikke en beskrivelse av hva som må til i ECJ for at systemet skal fungere. Filen som styrer programmets flyt er Number.java (vedlegg A.5).

Det første som skjer når en starter er at programmet leser inn tallrekken som skal løses sammen med de fire svaralternativene fra tastaturet. Deretter starter selve evolusjonen. Denne er delt i fire steg:

Steg 1. Evolver tallrekken som om den er lineær (det vil si med fem treningseksempler). Avslutt dersom programmet finner et optimalt individ. Dersom ikke optimalt individ blir funnet i løpet av fem generasjoner, gå videre til steg 2.

Steg 2. Evolver tallrekken som om den er en flervariabelrekke (det vil si med fire treningseksempler). Avslutt dersom programmet finner et optimalt individ. Dersom ikke optimalt individ blir funnet i løpet av femten generasjoner, gå videre til steg 3.

Steg 3. Evolver tallrekken som om den er to-alternerende (det vil si med to treningseksempler). Avslutt dersom programmet finner et optimalt individ. Dersom ikke optimalt individ blir funnet i løpet av fem generasjoner, gå videre til steg 4.

Steg 4. Evolver tallrekken som om den er en flervariabelrekke (det vil si med fire treningseksempler). Avslutt dersom programmet finner et optimalt individ. Dersom

ikke optimalt individ blir funnet i løpet av syttifem generasjoner, avslutt evolusjonen og bruk det beste individet så langt til å regne ut tallrekkens syvende tall.

Vi har altså ikke et eget program for å finne ut hva slags rekketype en rekke er. Et slikt program hadde vært en fordel, noe kapittel 5.5 diskuterer. Det viser seg likevel at måte å gjøre det på over fungerer fint. Hvor mange generasjoner hvert steg skal kjøre er litt tilfeldig, men antallet som er brukt denne implementasjonen har vist seg å være et passende antall.

Når evolusjonen er ferdig skriver programmet ut resultater fra kjøringen. Det beste individet fra kjøringen brukes til å finne ut hvilket tall programmet tror er det syvende i rekken. Dette skrives ut før man finner ut hvilket svaralternativ som ligger nærmest, og skriver ut dette også.

5.4 Resultater

Det implementerte systemet har forsøkt å løse tjuesyv tallrekkeoppgaver, og hver av oppgavene er forsøkt løst ti ganger. Oppgavene er forsøkt løst uten svaralternativer, det vil si at utdata fra systemet kun har gjettet på verdien til det syvende tallet i tallrekken. Vedlegg C inneholder detaljerte resultater fra kjøringene.

Vedlegget viser at systemet løser tallrekkeoppgaver meget godt. Av totalt 270 kjøringer fant programmet feil svar kun fire ganger. Det var to rekker som feilet, og begge feilet to ganger hver. Alle lineære rekker og alle flervariabelrekker ble løst riktig alle gangene, mens det var to to-alternerende rekker som feilet under kjøring.

Problemet med to-alternerende rekker er at de, som nevnt tidligere, kun inneholder to treningseksempler når rekken består av seks tall. Som vi skal se i figurrekkeimplementasjonen i kapittel 6 hender det også der av og til at noen av rekkene med kun to treningseksempler feiler. Vi kan dermed konkludere med at systemet yter meget godt i de tilfellene vi har tilgang til flere enn to treningseksempler, og over forventet i de andre tilfellene.

5.5 Mulige forbedringer

Det kan gjøres en rekke forbedringer med implementasjonen som mangler før systemet vil gjøre det optimalt når IQ-tester skal løses. Dersom implementasjonen slik den nå foreligger skal løse en IQ-test må en person manuelt se på IQ-testen og skrive inn tallrekken. Dette er naturligvis ikke ønskelig, da alt dette helst skal skje automatisk.

Det er derfor en nødvendighet at implementasjonen blir utvidet slik at:

- Oppgaven kan skannes, og teksten gjøres lesbar for maskinen.
- Oppgaveteksten forstås, slik at maskinen vet hva oppgaven går ut på.

Hvordan man kan oppfylle ønskene over er beskrevet i kapittel 3.2.4. Når disse ønskene er oppfylt vil implementasjonen være godt i stand til å løse oppgaver like dem gitt i Test nasjonen 2003 [NRK03] og 2004 [NRK04c]. Vi kan likevel ønske oss noen forbedringer av implementasjonen.

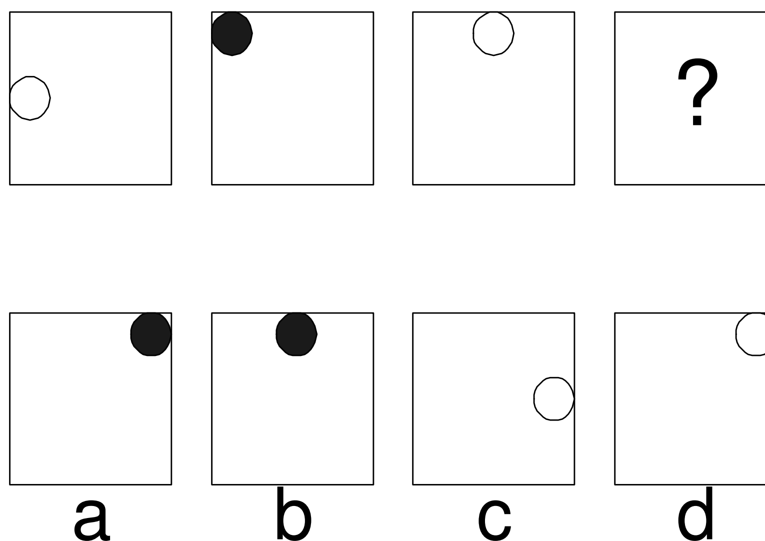
Spesielt vil det være en fordel dersom lengden på tallrekken ikke er begrenset til å være nøyaktig seks tall og at ikke antall svaralternativer er begrenset til nøyaktig fire. Fordi implementasjonen skal vise noen viktige prinsipper og dermed ikke trenger å være så generell som mulig har dynamisk lengde på tallrekke og svaralternativer ikke vært prioritert. De eneste forskjellene vil antageligvis være at alternerende rekker ikke bare vil være to-alternerende (kapittel 5.2.2), men også tre- og fire-alternerende når lengden på rekken som skal løses blir lang, og at hvert tall i flervariabelrekker (kapittel 5.2.3) vil kunne være avhengig av flere enn bare to av de foregående tallene. Oppgaver med veldig lange rekker vil dog sjeldent forekomme i IQ-tester, simpelthen fordi de vil oppleves som for vanskelige.

Et program for raskt å kunne kategorisere hva slags rekketype (lineær rekke, alternerende rekke, flervariabelrekke eller "jukserekke") er sterkt ønskelig, men finnes antageligvis ikke. Utfordringen sendes herved videre til dyktige matematikere.

Kapittel 6

Figurrekker (implementasjon)

Det finnes en rekke ulike varianter av oppgaver i IQ-tester som inneholder rekker av figurer. Hver figur består av et eller flere objekter, der det skjer en liten endring av objektene mellom de ulike figurene. Figur 6.1 [NRK03, oppgave 27] inneholder et eksempel på hvordan en slik oppgave kan se ut. Oppgaven består av en rekke med fire figurer der den siste er ukjent, og skal erstattes med et av svaralternativene a–d.



Figur 6.1: Figurrekke som skal løses ved hjelp av implementasjonen.

Dette kapitlet inneholder en oversikt over et implementert system for å løse slike figurrekker. Kapittel 6.1 inneholder en eksempeloppgave som skal løses ved hjelp av det implementerte systemet, og andre forutsetninger og krav som skal gjelde. Kapittel 6.2 inneholder en oversikt over hvordan objektene kan endre seg fra figur til figur, og kapittel 6.3 inneholder en beskrivelse av systemet som er implementert.

Til slutt, i kapittel 6.4, finnes en liten oversikt over resultater etter kjøring av implementasjonen, før mulige forbedringer til systemet diskuteres i kapittel 6.5.

6.1 Forutsetninger og krav

Implementasjonen skal løse oppgaven gitt i figur 6.1. Systemet skal også være generelt nok til å kunne løse oppgaver som ligner på oppgaven i figuren, men det forventes ikke at systemet skal kunne løse et vidt spekter av figurrekkeoppgaver.

I tillegg til å kunne løse oppgaven fra figur 6.1, finnes følgende krav til implementasjonen:

- Hver oppgave inneholder tre figurer, og oppgaven er å finne den fjerde.
- Til hver oppgave finnes fire løsningalternativer der kun ett er det riktige.
- Hver figur skal kun bestå av ett objekt, men implementasjonen skal enkelt kunne utvides til å støtte flere objekter per figur.
- Implementasjonen skal ikke ta hånd om uttrekning av egenskaper til objektene (dette ligger utenfor rapportens domene, se likevel kapittel 3.2.4.3 for mer om hvordan dette kan løses).

For enkelhetsskyld antar vi at figurene er 100 piksler brede og 100 piksler høye, og at sirklene i figurene har en diameter på 20 piksler. Vi antar også at x = 0, y = 0 er figurens midtpunkt, stigende koordinattall er mot henholdsvis høyre og oppover, og at økende rotasjonstall er *med* klokka.

I likehet med tallrekkeimplementasjonen (kapittel 5) skal all implementasjon foregå i Java. Teknikken som skal brukes er genetisk programmering (kapittel 4.1), og GP-rammeverket som skal benyttes er ECJ (kapittel 4.3).

6.2 Rekketyper

Det finnes et utall forskjellige rekketyper basert på visuelle serier [dSM94, side 14–28]. De ulike figurrekkene er ofte så vidt forskjellige at det ikke vil være mulig å gi en fullstendig oversikt, verken over hvilke rekketyper som finnes eller over hvordan de ulike rekketypene kan løses. Denne rapporten vil likevel prøve å gi et overblikk over de ulike typene.

Nokså vanlig er det å dele figuroppgavene i tre kategorier, bevegelse (kapittel 6.2.1), rotasjon (kapittel 6.2.3) og forandring (kapittel 6.2.3) [dSM94, side 12–13, 29]. Det går likevel ikke an å kategorisk putte en oppgave i én av de ovennevnte grupper. Ofte består

hver figur av flere ulike objekter, der hvert objekt både kan bevege seg, rotere og forandre seg uavhengig av de andre objektene. Dette gjør det meget komplisert å generalisere figurrekkeoppgavene.

6.2.1 Bevegelse

I mange oppgaver beveger noen av objektene seg fra en figur til en annen, vannrett, loddrett eller diagonalt, eller som følge av rotasjon rundt midten av figuren. Helt enkle bevegelser er enkle å følge. Vi trenger kun å vite x- og y-koordinatene til de ulike objektene i en figur og drive evolusjon på disse (en evolusjon som er lik evolusjonen av veldig enkle tallrekker fra kapittel 5).

Problemene oppstår når vi ikke har fullstendig informasjon om x- og y-koordinatene til alle objektene i figurene. Disse problemene oppstår dersom et objekt er utenfor figuren (et objekt kan for eksempel bevege seg fra å være innenfor figuren til å gå ut av den), eller dersom et objekt er skjult av en annen figur.

Dersom et objekt vises på kun to av de tre figuren må vi derfor anta at objektet også skulle vært på den tredje figuren. For å finne hvor i figuren (eller x- og y-koordinater utenfor figuren) må vi anta at objektet følger en lineær bevegelse eller at en eventuell rotasjon foregår med likt antall grader mellom hver figur. Så lenge vi bare ser et objekt på to av tre figurer vil ingen andre antagelser fungere.

For å observere en figurs bevegelse trenger vi med andre ord kun x- og y-koordinatene. For å finne ut om et objekt er rotert om en figurs midtpunkt må vi regne ut hvor objektets midtpunkt er i forhold til aksen som strekker seg fra figurens midtpunkt og rett oppover. Vi trenger dermed også figurens bredde og høyde.

For å regne ut et objekts plassering etter rotasjon om figurens midtpunkt brukes polare koordinater. Java har en egen metode som regner ut theta-verdien fra rektangulære koordinater, og vi bruker derfor denne.

6.2.2 Rotasjon

Ofte roterer en eller flere av objektene i en figur om seg selv. Den enkleste måten å finne ut hvor mye en figur er rotert er å legge to figurer oppå hverandre og finne ut hvor mange grader den ene figuren må roteres for at figurene skal dekke hverandre.

Dersom vi ikke kan finne ut hvor mye en figur er rotert i forhold til andre må vi sammenligne objektet på de to andre tilgjengelig figurene og anta at rotasjonen foregår med et konstant antall grader. Dette vil være det beste gjettet på hvor stor rotasjonen er fordi nesten alle figurer som roterer om seg selv gjør dette med konstant antall grader [dSM94, side 29].

For å ta høyde for rotasjon trenger vi dermed en variabel i implementasjonen som tar vare på hvor mange grader en figur er rotert i forhold til det tilsvarende objektet i første figur i figurrekken. Dersom objektets plassering skal beskrives ved hjelp av rotasjonen om figurens midtpunkt trenger vi også en variabel for objektets avstand fra figurens midtpunkt, noe følgende formel regner ut:

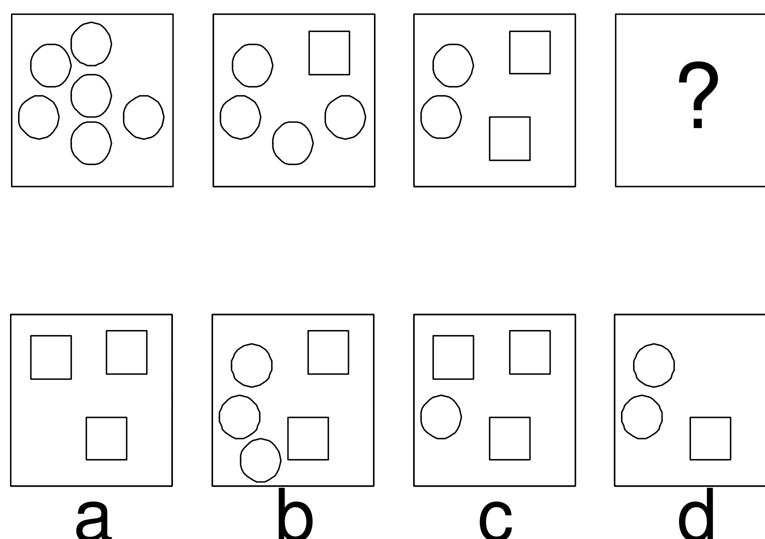
$$midtavstand = \sqrt{x^2 + y^2}$$

6.2.3 Forandring

I tillegg til bevegelse og rotasjon hender det at figurer forandrer seg. Disse forandringen inkluderer blant annet bytte av farge (som regel fra svart til hvit eller motsatt), endring av størrelse, endring av form og endring av antall.

For å ta vare på fargen til en figur trenger vi en variabel som representerer denne, og for å finne endring av størrelse bruker vi variablene fra kapittel 6.2.1 for å representere en figurs høyde og bredde.

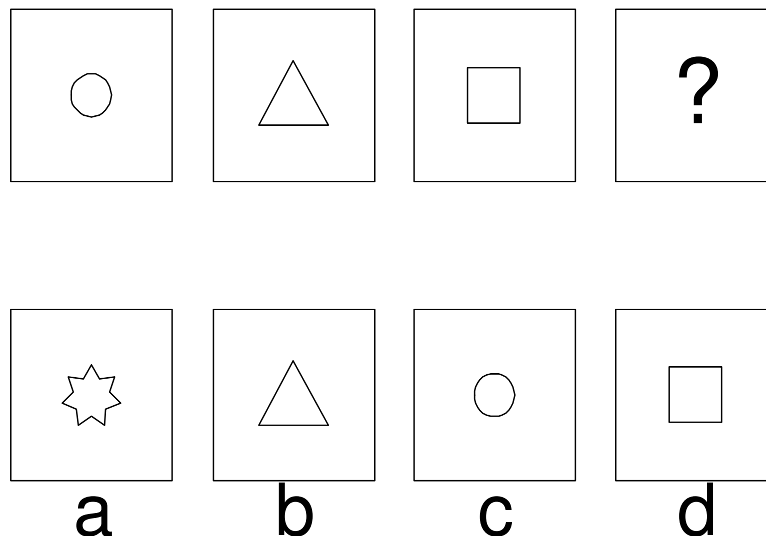
I enkelte oppgaver kan objektene endre både form eller antall. For eksempel kan det tenkes at en strek kan bli til en trekant, en trekant til en firkant og en firkant til en femkant, eller det kan tenkes at mellom figurene i oppgaven øker antall sirkler med én for hvert steg. Vi kan også tenke oss oppgaver der objektene endrer både form og antall. For eksempel kan *to* sirkler bli til én firkant, slik som figur 6.2 viser (i figuren er a det riktige svaralternativet).



Figur 6.2: Oppgave der *to* sirkler i en figur blir til én firkant i neste.

Ettersom implementasjonen ikke tar hensyn til figurer med flere enn ett objekt trenger vi ikke å tenke på det her. Dersom man likevel skulle implementert et system som tok hensyn til figurer med mange objekter måtte man telt hvor mange objekter som fantes av hver form, og evolvert dette tallet også. Dersom for eksempel antall sirker øker med ett for hver figur er det naturlig at den siste figuren har en sirkel mer enn den nest siste.

Hvis vi antar at vi har en rekke der hver figur bare har et objekt, og objektene i de tre første figurene i rekken er ulike, vil også objektet i den fjerde figuren være ulik objektene i de tre forrige. I figur 6.3 er svaralternativ a det riktige (det er nokså vanlig å gjøre det på denne måten i IQ-tester uten at det nødvendigvis er så veldig logisk). I oppgaver der objektene i første og tredje figur er like, vil også objektene i andre og fjerde være det. Oppgaver der objektene har ulike former vil ikke bli løst av implementasjonen.



Figur 6.3: Oppgave der formen på objektet endrer seg mellom hver figur.

6.3 Beskrivelse av implementert system

Det er laget et system for å løse oppgaven beskrevet i kapittel 6.1. I dette kapitlet finnes en oversiktlig beskrivelse av hvordan systemet er implementert. Kildekoden til systemet finnes på den vedlagte CD-en og i vedlegg D, og et kjøreeksempel finnes i vedlegg E.

Som nevnt i kapittel 4 er systemet skrevet i Java, genetisk programmering er brukt til å løse problemet, og GP-rammeverket som er brukt er ECJ.

6.3.1 Objektenes egenskaper

Som vi så i kapittel 6.2 trenger vi en rekke variabler for å beskrive hvert enkelt objekt i figuren. Tabell 6.3.1 inneholder en oversikt over hvilke variabler som er brukt i denne implementasjonen, hva slags datatyper hver av variablene kan han, og en kort beskrivelse av variablene.

Tabell 6.1: Variabler i figurrekke-implementasjonen.

Variabel	Datatype	Beskrivelse
x	Heltall	X-koordinat til objektets midtpunkt.
y	Heltall	Y-koordinat til objektets midtpunkt.
bredde	Heltall	Objektets bredde.
høyde	Heltall	Objektets høyde.
figurrotasjon	Heltall	Objektets rotasjon om figurens midtpunkt i antall grader ($0 \leq$ figurrotasjon < 360).
midtavstand	Heltall	Objektets avstand fra figurens midtpunkt.
farge	Heltall	Objektets farge der 0 = sort og 1 = hvit.
objektrotasjon	Heltall	Objektets rotasjon om seg selv i antall grader ($0 \leq$ objektrotasjon < 360) målt fra første figur i figurrekken.

Som nevnt i kapittel 6.2.3 vil ikke oppgaver der objektenes form er ulike bli løst av implementasjonen. Vi har derfor ingen parameter til beskrive objektets form. Ettersom implementasjonen kun skal tillate ett objekt per figur trenger vi ikke ta hensyn til hvilke objekter i ulike figurer som tilsvarer hverandre.

6.3.2 Evaluering av egenskapene

Kapittel 6.3.2.1, 6.3.2.2, 6.3.2.3 og 6.3.2.4 forklarer hvordan de ulike egenskapene fra tabell 6.3.1 evalueres.

6.3.2.1 X og Y. Figurrotasjon og midtavstand

Det finnes flere måter å representere et objekts posisjon på. Den vanligste er ved hjelp av rektangulære koordinater (det vil si x- og y-koordinater), mens polare koordinater (det vil si ved hjelp av avstanden fra figurens sentrum og antall grader fra linjen loddrett opp fra figurens sentrum) er en mindre vanlig måte å representere et objekts posisjon på. Objektet er i figurens midtpunkt dersom $x = 0$ og $y = 0$, og dersom $midtavstand = 0$.

I implementasjonen prøver vi derfor med evolusjon både å finne ut om et objekt beveger seg langs x- og/eller y-aksen eller om det roterer om figurens sentrum. Systemet oversetter derfor rektangulære koordinater (x og y) til polare koordinater (figurrotasjon og midtavstand). Deretter justeres figurrotasjonen på samme måte som objektrotasjonen (se kapittel 6.3.2.4 for algoritme).

Som sagt blir både x og y, og figurrotasjon og midtavstand evolvert. For alle parametrene undersøkes det om første og tredje er like. Dersom de er det vil fjerde parameter være lik andre. Dersom første og tredje parameter derimot er ulike starter evolusjonen for å finne x og y, og figurrotasjon og midtavstand for den fjerde figuren i rekken. Dersom systemet etter evolusjonen ikke klarer å finne x eller y brukes figurrotasjon og midtavstand til å finne fjerde figur i rekken (dette er tilfellet for oppgaven i figur 6.1), mens dersom systemet etter evolusjonen ikke klarer å finne figurrotasjon eller midtavstand brukes x og y til å finne fjerde figur.

Dersom systemet imidlertid klarer å evolvere seg frem til både x og y, og figurrotasjon og midtavstand må vi utføre en sammenligning for å finne ut hvilken som er best. Vi har altså funnet følgende:

- $f(x_i) = x_{i+1}$
- $f(y_i) = y_{i+1}$
- $f(figurrotasjon_i) = figurrotasjon_{i+1}$
- $f(midtavstand_i) = midtavstand_{i+1}$

For hvert av de fire svaralternativene i oppgaven prøver vi deretter å finne ut hva som blir minst av

$$|x_4 - x_{svaralt.\ i}| + |y_4 - y_{svaralt.\ i}|$$

og

$$|figurrotasjon_4 - figurrotasjon_{svaralt.\ i}| + |midtavstand_4 - midtavstand_{svaralt.\ i}|$$

Dersom sistnevnte er minst for svaralternativ i velger vi å bruke figurrotasjon og midtavstand til å representere et objekts plassering, ellers bruker vi x- og y-koordinater.

Legg merke til at det implementerte systemet alltid krever at man kun skriver inn x- og y-koordinater når programmet starter, mens svaret kun blir skrevet ut ved hjelp av figurrotasjon og midtavstand, uavhengig av hvilken metode evolusjonen fant ut var den beste til å representere objektets plassering på.

6.3.2.2 Bredde og høyde

Bredde og høyde evalueres på samme måte, og i to varianter. Dersom bredden/høyden på objektet i første figur er lik bredden/høyden på objektet i tredje figur vil bred-

den/høyden på objektet i fjerde figur være lik bredden/høyden på objektet i andre figur.

Dersom bredden/høyden på objektet ikke er lik i første og tredje figur evolveres egenskapen på samme måte som tallrekkene gjorde i kapittel 5. Vi skal dermed prøve å finne følgende:

- $f(objektbredde_i) = objektbredde_{i+1}$

Eventuelt:

- $f(objekthøyde_i) = objekthøyde_{i+1}$

Rekken har kun to treningseksempler, nemlig:

- $f(objektbredde_1) = objektbredde_2$
- $f(objektbredde_2) = objektbredde_3$

Eventuelt:

- $f(objekthøyde_1) = objekthøyde_2$
- $f(objekthøyde_2) = objekthøyde_3$

6.3.2.3 Farge

Når vi kun har to farger finnes det bare to logiske fargerekker:

- Alle objektene har samme farge.
- Objektene veksler mellom å være hvite og sorte.

Vi kan dermed konkludere med at objektet i den fjerde figuren må ha samme farge som objektet i den andre figuren. Dersom det viser seg at fargene ikke følger noen av de to mønstrene ovenfor har vi likevel ikke noe annet gjett som er bedre (fordi ingen andre fargekombinasjoner er logiske).

6.3.2.4 Objektrotasjon

Etter at parametrene til objektrotasjon er lest inn må noen disse justeres (dette for at evolveringen skal fungere riktig). Justering må finne sted dersom den midterste av de tre parametrene er lavere eller høyere enn begge de to andre og første parameter ikke er lik tredje. Anta for eksempel at det første objektet er rotert 0 grader om seg selv, det andre er rotert 270 grader om seg selv, og det tredje er rotert 180 grader om seg selv. Vi velger dermed å øke den første parameteren med 360 grader. På denne måten klarer evolusjonen lett å finne at den fjerde parameteren må være 90 grader.

Algoritmen for å justere objektrotasjonsparameterene (or1, or2 og or3) er som følger:

- Hvis or1 ikke er lik or3, gjør følgende:
  - Hvis or2 $<$ or1 og or2 $<$ or3, gjør følgende:
    - Hvis or1 $>$ or3, øk or2 og or3 med 360.
    - Ellers øk or1 og or2 med 360.
  - Hvis or2 $>$ or1 og or2 $>$ or3, gjør følgende:
    - Hvis or1 $>$ or3, øk or3 med 360.
    - Ellers øk or1 med 360.

Deretter evolveres objektrotasjonen på samme måte som bredde og høyde gjør (kapittel 6.3.2.2): Dersom første og tredje parameter er like er fjerde parameter lik andre. Ellers skjer evolusjonen på samme måte som tallrekkene i kapittel 5.

6.3.3 GP-funksjoner, -konstanter og -variabler

Systemet består av tre funksjoner, tre konstanter og en variabel. Generelt om funksjoner, konstanter og variabler finnes i kapittel 4.1.1.

Systemet tre funksjoner er:

Add.java Legger sammen to tall.

Mul.java Multipliserer to tall.

Sub.java Trekker et tall fra et annet.

Systemet tre konstanter er:

One.java Heltallet 1.

Ten.java Heltallet 10.

Fourtyfive.java Heltallet 45.

Konstanten One er med for å kunne evolvere med tall der forskjellene er små, mens Ten er med for å slippe å bruke One veldig mange ganger når tallene blir noe større. Fourtyfive er med fordi 45, og tall multiplisert med 45, er vanlige gradstall.

Systemet ene variabel er:

X.java

6.3.4 GP-parametre

Alle GP-parametrene brukt i implementasjonen finnes i såkalte paramter-filer. Med to unntak har alle GP-parametrene brukt i forbindelse med implementasjonen de verdiene som er standard for ECJ (de viktigste av disse diskutert i kapittel 4.3.2.1). De to parametrene som ikke har standardverdier er:

Elitisme De 3 beste individene fra hver generasjon går videre til neste. Dette sikrer blant annet at uansett når evolusjonen stopper vil de tre beste individene totalt sett finnes i den siste generasjonen.

Generasjoner. Antall generasjoner er satt til 1000 slik at vi er sikker på at evolusjonen blir ferdig.

Generatoren som genererer tilfeldige tall blir sådd med tid slik at hver kjøring blir tildelt tall ulik de forrige kjøringene.

6.3.5 Systemets virkemåte

Dette kapitlet inneholder en beskrivelse av den delen av systemet som er et tillegg til GP-rammeverket (ECJ) fungerer, og altså ikke en beskrivelse av hva som må til i ECJ for at systemet skal fungere.

Det første som skjer når en starter er at programmet leser inn parametre om figurrekken og svaralternativene fra tastaturet. Deretter regnes figurrotasjon og midtavstand ut (kapittel 6.3.2.1) før objektrotasjon og figurrotasjon justeres som beskrevet i henholdsvis kapittel 6.3.2.4 og kapittel 6.3.2.1. Systemets røde tråd finnes i filen Figures.java (vedlegg D.3).

Deretter starter selve evolusjonen. Denne er delt i ni steg (et steg for hver av de åtte parametrene og et steg for å skrive ut resultatene fra kjøring). De ni stegene er som følger:

Steg 1–8. Evaluer parametrene fra tabell 6.3.1 slik som beskrevet i kapittel 6.3.2.1–6.3.2.4. Evalueringen er delt i tre:

1. Dersom første og tredje parameter er like settes fjerde parameter lik andre, og evalueringen av parameteren er ferdig.

2. Hvis ikke starter en vanlig evolusjon med to treningseksempler (se kapittel 6.3.2.2 for eksempel). Hvert individs fitnessverdi (kapittel 4.1.2) regnes ut, og når et optimalt individ er funnet avsluttet evolusjonen av parameteren. Deretter brukes det optimale individet til å regne ut verdien til parameteren i den fjerde figuren i figurrekken.

3. Fordi vi kun har begrenset antall tid på å løse en figurrekkeoppgave begrenser vi antall evolveringen av hver parameter til å gjelde fem tusen individer. Dersom det da ikke er funnet et optimalt individ har ikke programmet klart å finne verdien til parameteren, vi begynner derfor å evaluere neste parameter i stedet.

Steg 9. Når alle åtte parameterne er evaluert er det klart for å skrive ut resultatene. For parametrene bredde, høyde, farge og objektrotasjon skrives resultatene ut dersom programmet har klart å finne en verdi til parameteren. For de parametrene systemet eventuelt ikke har funnet noen verdi til skrives dette også ut.

Hvorvidt vi skal bruke x og y, eller figurrotasjon og midtavstand til å beskrive et objekts plassering i figuren blir bestemt slik som beskrevet i kapittel 6.3.2.1.

Vi trenger dermed seks parametre til å beskrive den fjerde figuren i figurrekken, nemlig bredde, høyde, farge og objektrotasjon i tillegg til to parametre for å bestemme objektets plassering. For å finne ut hvilket svaralternativ som er det riktige teller vi hvor mange av de utregnede parametrene som stemmer for hvert av svaralternativene. For eksempel får et svaralternativ fire poeng dersom kun bredde, høyde, farge og objektrotasjon stemmer. Det svaralternativet som får flest poeng (seks poeng er maksimum) antar vi er det riktige svaralternativet.

6.4 Resultater

Det implementerte systemet har forsøkt å løse fem figurrekkeoppgaver, og hver av oppgavene er forsøkt løst fem ganger. Oppgavene er forsøkt løst uten svaralternativer, det vil si at utdata fra systemet kun har vært de ferdig evolverte parameterne. Vedlegg F inneholder detaljerte resultater fra kjøringene.

Systemet forsøker å finne totalt åtte parametre ved hver kjøring. Objektets bredde, høyde, farge og objektrotasjon evalueres hver for seg, mens for parametrene som beskriver et objekts plassering evalueres to par med parametre der begge parene beskriver objektets posisjon.

Av i alt tjuefem kjøringer (fem oppgaver á fem kjøringer) ble kun to parametere evaluert feil. Grunnen til at resultatene var såpass gode var at parametrene danner grunnlaget for enkle rekker, rekker som er mye enklere enn dem vi behandlet i kapittel 5. Det er for eksempel ikke vanskelig å finne ut bredden til det fjerde objektet i rekken dersom bredden til de tre første objektene var like.

Rekkene med parametre er som sagt enkle. De faller nesten alltid innenfor en av følgende kategorier:

- Alle parametrene er like.

- Annenhver parameter er lik (dette er spesielt vanlig for farger).
- Parametrene stiger eller synker lineært.

De to første punktene tar systemet hensyn til, og dersom en parameter er innenfor en av de to kategoriene starter ikke evolusjonen (på den parameteren). Det hadde antageligvis vært en fordel å, i de tilfellene der parametrene stiger eller synker lineært, ikke starte evolusjon, men bare regne ut verdien til den siste parameteren. Dette hadde vært en raskere måte å gjøre det på enn å evolvere seg frem. Evolusjonen kunne da ha startet kun på parametre som ikke ikke var i en av de tre ovennevnte kategoriene (altså kun på et lite fåtall).

6.5 Mulige forbedringer

Det bør gjøres en rekke forbedringer av implementasjonen som mangler før systemet vil gjøre det optimalt når IQ-tester skal løses. Dersom implementasjonen slik den nå foreligger skal løse en IQ-test må en person manuelt se på IQ-testen og skrive inn parametrene som tilsvarer hvert objekt i figurene. Dette er naturligvis ikke ønskelig, da alt dette helst skal skje automatisk.

Det er derfor en nødvendighet at implementasjonen blir utvidet slik at:

- Oppgaven kan skannes, og at teksten og figurene i oppgaven kan skilles fra hverandre.
- Oppgaveteksten forståes, slik at maskinen vet hva oppgaven går ut på.
- Figurene, og objektene i figurene, automatisk blir representert med nødvendige parametre (blant annet de fra tabell 6.3.1).

Hvordan man kan oppfylle ønskene over er beskrevet i kapittel 3.2.4. Men selv om disse ønskene oppfylles vil fortsatt kun et lite fåtall av oppgavene bli løst korrekt. Det er hovedsaklig to problemer som gjenstår, nemlig at hver figur kan inneholde flere enn ett objekt, og at et objekt kan endre form fra én figur til en annen.

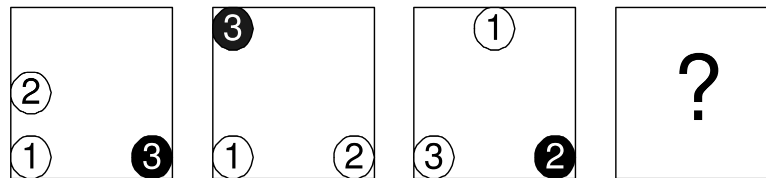
Problemet med oppgaver som inneholder flere enn ett objekt per figur er at det er vanskelig å finne ut hvilket objekt i en figur som tilsvarer et annet objekt i en annen figur. Én måte er å sammenligne hvert objekt i hver figur med hverandre. Anta at vi har tre objekter per figur, slik som i figur 6.4 (tallene i figurene er ikke en del av oppgaven, men kun for å kunne henvise til de ulike objektene).

Vi lager deretter en tabell over alle kombinasjoner av alle objekter, slik som tabell 6.5 viser et utsnitt av. Første kolonne i tabellen viser hvilke objekter vi sammenligner. For eksempel betyr 2 1 1 at vi sammenligner objekt 2 i første figur med objekt 1 i andre figur

og objekt 1 i tredje figur. Dersom en parameter er lik for alle de tre objektene setter vi en x i tabellen, hvis ikke setter vi en o.

Når hele tabellen er fylt ut finner vi ut hvilke tre rader (der alle tre objektene i alle tre figurer er representert, for eksempel rad 1 1 2, rad 2 2 3 og rad 3 3 1) som til sammen inneholder flest x-er. Vi antar at dersom tre objekter er på samme rad (blant de tre radene som er trukket ut), hører disse objektene sammen (for eksempel objekt 1 i første figur, objekt 1 i andre figur og objekt 2 i tredje figur dersom radene som er trukket ut er som over).

Det kan tenkes at noen av parametrene bør vektes høyere enn de andre i utvelgelsen. Blant annet er det lite vanlig med mange forandringer i form når antall objekter i en figur er mange.



Figur 6.4: Oppgave med flere objekter per figur.

Tabell 6.2: Sammenligning av parametre til ulike objekter.

Objekter	**X**	**Y**	**Bredde**	**Høyde**	**Farge**	**Objektrotasjon**	**Form**
1 1 1	o	o	x	x	x	x	x
1 1 2	o	x	x	x	o	x	x
1 1 3	x	x	x	x	x	x	x
2 1 1	o	o	x	x	o	x	x
...	...	...	...	...	...	...	...

I rekker der objektene endrer form er det nokså vanskelig å finne ut hvilken form som er den riktige. Et par ting går det likevel an å si. I oppgaver der alle objektene er ulike (slik som i figur 6.3), må vi anta at objektet i den siste figuren er ulik de andre objektene.

I tillegg er det vanlig at figurrekker der objektene øker i kompleksitet (for eksempel der strek blir til trekant, trekant til firkant, firkant til femkant, og så videre), fortsetter å øke i kompleksitet, og at figurrekker der objektene minsker i kompleksitet fortsetter å minske.

Figurrekker der objektene endrer form er likevel så komplekse å løse at en rapport som denne bare vil kunne se på en liten del av hele problemet.

Kapittel 7

Andre oppgavetyper

Dette kapitlet inneholder en oversikt over alle oppgavetypene nevnt i kapittel 3.2.2. For hver oppgavetype følger en eksempeloppgave med grundig beskrivelse av hvordan denne kan løses ved hjelp av en datamaskin. I tillegg finnes det for de oppgavetypene der det er relevant en diskusjon av hvordan lignende oppgaver kan løses.

For alle oppgavene antar vi at innskanning (kapittel 3.2.4.1), naturlig språk-forståelse (kapittel 3.2.4.2) og kunnskapsrepresentasjon (kapittel 3.2.4.3) er i orden. Det antaes blant annet at naturlig språk-forståelsen er så god at datamaskinen "forstår" hva oppgaven går ut på. I noen oppgaver kreves det ekstra mye av naturlig språk-forståelsen og kunnskapsrepresentasjon, og disse kommenteres derfor spesielt for enkelte av oppgavene.

7.1 Språk

God språkforståelse er en forutsetning for å gjøre det bra på oppgavene i denne kategorien. Kategorien er delt i tre. Første oppgavetype går ut på å finne hvilket ord som ikke passer inn blant de øvrige (kapittel 7.1.1), mens andre oppgavetype går på å sortere bokstaver til å danne ord (kapittel 7.1.2). Tredje oppgavetype går ut på å sammenligne ulike ordspråk med hverandre (kapittel 7.1.3), mens fjerde og siste oppgavetype går ut på å finne synonymer, altså ord som betyr det samme (kapittel 7.1.4).

7.1.1 Hvilket ord passer ikke inn?

Å finne hvilket ord i en liste som ikke passer med de øvrige er en vanlig oppgave i IQ-tester, og eksempeloppgaven er derfor godt representativ for en rekke oppgaver.

7.1.1.1 Eksempeloppgave

Oppgave (hentet fra Test nasjonen 2004, samme kilde forteller at man har ti sekunder på å løse oppgaven [NRK04c, oppgave 1]):

> Hvilket ord passer ikke?
> a) enkelt
> b) lett
> c) ukomplisert
> d) vektløs

Når et av ordene ikke passer inn betyr det at de tre andre ordene må henge sammen på en eller annen måte. Å finne ut hvordan ordene henger sammen er ikke alltid like. Det beste vi kan gjøre er å oversette ordene til engelsk og deretter slå opp hvert av dem i WordNet [Uni05]. Alle ord i WordNet er godt beskrevet og vi kan derfor håpe på at vi ved hjelp av disse beskrivelsene kan finne ut at noen av ordene henger sammen (det kan for eksempel tenkes at et av ordene er med på å beskrive et av de andre).

Dersom ikke WordNet kan hjelpe oss kan vi prøve samme strategi med KnowItAll [oW05]. Mer om WordNet og KnowItAll finnes i kapittel 3.2.3.

7.1.1.2 Hvordan løse lignende oppgaver

Det finnes få oppgavetyper som ligner på eksempeloppgaven uten å være lik. Å finne hvilket ord i en liste som betyr det samme som et gitt ord er oppgavetypen som ligner mest. Denne disse oppgavene er enklere, i alle fall for datamaskinen, fordi en god synonymordbok vil i de fleste tilfeller vil gjøre jobben. Se kapittel 7.1.4 for hvordan slike oppgaver løses.

7.1.2 Sortere bokstaver til å danne ord

Dette kapitlet inneholder en beskrivelse av hvordan man løser oppgaver som krever at man sorterer ulike bokstaver til å danne ord.

7.1.2.1 Eksempeloppgave

Oppgave (hentet fra Test nasjonen 2004, samme kilde forteller at man har tjue sekunder på å løse oppgaven [NRK04c, oppgave 6]):

> Hva blir en sportsgren etter riktig sortering?
> a) ROMST
> b) NEJLI

c) DEKOK
d) YALRL

Vi antar at innskanningen, tekstgjennkjenningen og naturlig språk-forståelse ikke er en nevneverdig utfordring (dersom den skulle være det, se henholdsvis kapittel 3.2.4.1 og kapittel 3.2.4.2). Vi kan dermed forutsette at datamaskinen forstår oppgaven.

Vi skal finne ut hvilket løsningsalternativ som er slik at bokstavene kan stokkes til en sportsgren. Følgende løsningsmetode vil fungere:

- Oversett ordet sportsgren til engelsk.
- Finn, ved hjelp av WordNet, alle hyponymene til sportsgren, oversett disse til norsk, og ta vare på de som på norsk har nøyaktig fem bokstaver (ettersom om alle ordene i oppgavene har fem bokstaver).
- For hvert løsningsalternativ, gjør følgende:
  - Sjekk om noen av de oversatte ordene inneholder nøyaktig de samme bokstavene som ordet. Dersom de gjør det har vi funnet løsningen på oppgaven.

Ved hjelp av denne løsningsmetoden vil, forhåpentligvis, datamaskinen stokke bokstavene i alternativ d til ordet RALLY og finne ut at dette er en sportsgren.

7.1.2.2 Hvordan løse lignende oppgaver

Det finnes få, men like varianter av eksempeloppgaven i forrige kapittel. For eksempel kunne oppgaven over spurt om hvilket ord som *ikke* ble en sportsgren ved stokking av bokstavene. Eventuelt kunne oppgaven spurt om et hyponym i stedet for et hypernym. Oppgaven vil likevel ligne så mye på eksempeloppgaven at det ikke bør være noe problem å også implementere variantene nevnt her.

7.1.3 Ordspråk

Det finnes flere hundre norske ordspråk. Ordspråkene har til felles at de ikke skal taes bokstavlig. De har altså en annen og mer dypere betydning enn slik de står. Mange IQ-oppgaver består i å finne ut av hva et ordspråk faktisk betyr, eller finne ut hvilke ordspråk som betyr det samme.

7.1.3.1 Eksempeloppgave

Oppgave (hentet fra Test nasjonen 2004, samme kilde forteller at man har tjuefem sekunder på å løse oppgaven [NRK04c, oppgave 12]):

> Hvilket ordspråk har liknende betydning som dette?
> Den som ingenting har, har ingenting å miste
> a) Penger er ikke alt
> b) Pengelaus mann går trygt mellom tjuvar
> c) Tom pung gir aktsom mann
> d) Fattig og rik gjør døden lik

Disse oppgavene er veldig vanskelige, for ikke si umulige å løse. Noe av det som gjør oppgavene vanskelige å løse er at et ordspråk betyr noe annet enn det som egentlig står der. Det vil derfor ikke være mulig å sammenligne ordspråkene direkte mot hverandre for å se hvilke som er like.

Den eneste måten å løse en slik oppgave på må være dersom det allerede finnes en oversikt der ordspråk er kategorisert etter lik betydning. Finnes en slik oversikt er oppgaven enkel å løse. Fordi oppgaven ikke er kulturfri vil den ikke bli vektlagt stort i rapporten (se forøvrig kapittel 3.2.1 om kritikk av IQ-oppgaver).

7.1.4 Synonymer

To ord er synonymer dersom de betyr det samme og oppgaver basert på synonymer er, som vi skal se, vanligvis relativt enkle for datamaskinen å løse.

7.1.4.1 Eksempeloppgave

Oppgave (hentet fra Test nasjonen 2003, samme kilde forteller at man har ti sekunder på å løse oppgaven [NRK03, oppgave 6]):

> Hvilket ord har samme betydning?
> konflikt
> a) skille
> b) problem
> c) tvangssituasjon
> d) strid

Den beste strategien for å løse oppgaven er å slå opp det gitte ordet (i dette tilfellet ordet konflikt) i en synonymordbok (for eksempel Clue [ASA05]) og se om noen av de fire andre ordene er synonymt med det gitte ordet. Dersom et av ordene står i synonymordboken er oppgaven løst. Dersom ikke ordet står i synonymordboken kan vi gjøre som i kapittel 7.1.1 og se om et av ordene i oppgaven beskriver det gitte ordet.

7.1.4.2 Hvordan løse lignende oppgaver

Oppgavene som ligner mest går ut på å finne hvilket ord i en liste som ikke passer inn. Slike oppgaver er beskrevet i kapittel 7.1.1.

7.2 Hukommelse og konsentrasjon

Verken hukommelses- eller konsentrasjonsoppgaver er typiske IQ-oppgaver, men brukes ofte til å teste nettopp hukommelse og konsentrasjon. Fordi datamaskiner har god lagringskapasitet (i betydningen: kan både lagre mye data og hente frem mye lagret data på kort tid) og er svært nøyaktige scorer disse, som kapitlet viser, meget godt på slike oppgavetyper.

7.2.1 Hukommelse

Hukommelsesoppgaver er for datamaskinen tilsynelatende enkle oppgaver. Som vi skal se i de følgende kapitlene ligger den største utfordringen med hukommelsesoppgaver i å representere det som skal huskes på en god måte. Denne utfordringen er omtalt grundigere i kapittel 3.2.4.3, men vil med en eksempeloppgave og hvordan denne løses (kapittel 7.2.1.1), og hvordan lignende oppgaver løses (kapittel 7.2.1.2) bli omtalt spesielt for hukommelsesoppgaver i dette kapitlet.

7.2.1.1 Eksempeloppgave

Tabell 7.1: Hukommelse: Matrise med bokstaver.

S	F	E	R
O	U	M	T
C	W	K	P

Tabell 7.2.1.1 viser hvordan en hukommelsesoppgave kan se ut (hentet fra Test nasjonen 2004, samme kilde forteller at man har tretti sekunder på å se på matrisen med bokstaver, og deretter ti sekunder på å finne ut hvilken bokstav som ikke var med i matrisen [NRK04c, oppgave 18]), mens selve oppgaven er som følger:

Hva var ikke med?
a) K
b) T
c) O
d) B

Oppgaven har tre utfordringer:

1. Lagre matrisen med bokstaver på en slik måte at denne kan fremkalles senere (når det gjelder en matrise med bokstaver er denne utfordringen betydelig større for mennesker enn for datamaskiner).

   Fordi man ikke kan vite hva som er den egentlig oppgaven er det viktig at innholdet i matrisen lagres så likt som mulig, det vil blant annet si å lagre bokstavene i den rekkefølgen de er gitt. Dette er en enkel oppgave for en datamaskin, man kan ta et hvilket som helst programmeringsspråk og lagre bokstavene i en to-dimensjonal tabell der.

2. Forstå hva oppgaven spør etter (spør den for eksempel om en bokstav er *med* eller *ikke med*? I oppgaven over kan ikke dette regnes som en utfordring for mennesker, mens det for en datamaskin er nokså problematisk). Se kapittel 3.2.4.2 for hvordan dette kan løses.

3. Løse oppgaven. For eksempeloppgaven over vil det si å sammenligne hvert av svaralternativene med bokstavene i matrisen. Dersom man finner svaralternativet i matrisen prøver man neste svaralternativ helt til man har funnet det svaralternativet som ikke finnes i matrisen. Også dette er en meget enkel oppgave for en datamaskin.

Ved hjelp av stegene finner datamaskinen forhåpentligvis ut hvilken bokstav som ikke er representert i matrisen, nemlig alternativ d, B.

7.2.1.2 Hvordan løse lignende oppgaver

Hukommelsesoppgaver finnes i ulike varianter, og mange av dem ligner på eksempeloppgaven fra forrige kapittel. Nokså vanlig er lignende matrise med tall, ord eller figurer som skal huskes. Og huske tall og ord er, for en datamaskin, like vanskelig som å huske bokstaver.

Hukommelsesoppgaver med figurer er en noe mer komplisert prosess, men også disse bør i de fleste tilfeller kunne løses enkelt av en datamaskin. Ved å sammenligne to figurer punkt for punkt kan man anta at figurene er like dersom så og si alle punktene er like.

En annen variant av hukommelsesoppgaver er at en blir presentert for en tekst som man, etter å ha lest og gitt fra seg teksten, må svare på noen spørsmål til (se for eksempel oppgave 194 i Séréville og Myers bok om hvordan man får sukess i IQ-tester [dSM94, oppgave 194]). Slike hukommelsesoppgaver minner, for datamaskinen, mye om tekstlige regneoppgaver (kapittel 7.4).

I slike hukommelsesoppgaver ligger utfordringen for datamaskinen ikke i å lagre det som blir presentert for den, men å omgjøre naturlig språk (både setningene den skal

huske og spørsmålene som blir stilt til teksten etterpå) slik at datamaskinen faktisk blir i stand til å svare på spørsmålene. For en mer detaljert fremstilling av hvordan dette kan løses, se kapittel 3.2.4.2.

Dette kapitlet har vist at hukommelsesoppgaver (med unntak av oppgaver der man på svare må spørsmål til en tekst) er blant de letteste IQ-oppgavene en datamaskin kan bli stilt overfor, og kapitlet har forhåpentligvis gitt en pekepinn på hvordan man bør tenke hvis man skal løse hukommelsesoppgaver ved hjelp av en datamaskin.

7.2.2 Konsentrasjon

Mange yrker krever stor evne til konsentrasjon, og det er derfor populært å teste dette i forbindelse med jobbansettelser [dSM94, side 126], og mens enkelte yrker krever stor grad av presisjon til middels fart krever andre yrker høy fart til middels presisjon. Hvorvidt man derfor skal gjøre oppgaven sent og være sikker på å få alt riktig eller gjøre den fort og tillate seg noen feil blir derfor en vurderingssak. Som vi skal se i eksempeloppgaven beskrevet under har datamaskinen evnen til å løse konsentrasjonsoppgaver med både stor nøyaktighet og stor fart. Datamaskinen er derfor overlegen mennesker på slike typer oppgaver.

7.2.2.1 Eksempeloppgave

Oppgave (hentet fra Séréville og Myers bok om hvordan man får suksess i IQ-tester [dSM94, oppgave 187], samme kilde forteller at man har 125 sekunder på å løse oppgaven):

> Kryss av de forfatterne som ikke forekommer i begge kolonnene i tabell 7.2.2.1.

Oppgaven løses enkelt av en datamaskin ved hjelp av følgende løsningsstrategi:

1. For hvert navn i kolonnen til høyre, sjekk om navnet finnes i kolonnen til høyre. Slett navn som finnes i begge listene.
2. Når alle navnene er sjekket, sett kryss ved de navnene som er igjen i kolonnene (slik oppgaven krever).

7.2.2.2 Hvordan løse lignende oppgaver

I tillegg til oppgaver som er like eksempeloppgaven består konsentrasjonsoppgaver ofte i å sammenligne ord og tall for å se om disse er skrevet på samme måte. Slike oppgaver

Tabell 7.2: Konsentrasjonsoppgave.

Dickens	Zola
Grundtvig	Rimbaud
Sandemose	Andersen
Scherfig	Carroll
Lindgren	Scherfig
Pagnol	Colette
Doyle	Hugo
Zola	Dickens
Colette	Camus
Mitchell	Kierkegaard
Bjørnson	Bang
Dumas	Shakespeare
Rimbaud	Eco
Sand	Dumas
Shakespeare	Proust
Andersen	Sand
Proust	Giono
Camus	Pagnol
Garborg	Goethe
Kierkegaard	Lindgren
Goethe	Flaubert
Flaubert	Grundtvig
Carroll	Mitchell
Daudet	Jacobsen
Bang	Bjørnson

er enkle for datamaskinen. Man trenger kun å sjekke bokstav for bokstav og tall for tall for å se om disse er like. Dette er så enkelt at det ikke vil bli videre utdypet her.

7.3 Logikk

Logikkoppgavene er de oppgavene som vanligvis mest blir forbundet med IQ-oppgaver. I dette kapitlet vil fem ulike oppgavetyper innen logikk bli omtalt. Kapittel 7.3.1 omtaler figurrekker, og kapittel 7.3.2 omtaler tallrekker. Bokstavrekker minner mye om tallrekker, og blir omtalt i kapittel 7.3.3, mens figurmatrise-oppgaver ligner på figurrekke-oppgaver, og blir omtalt i kapittel 7.3.4. Til slutt, i kapittel 7.3.5, omtaler vi en oppgavetype kalt ordpar, der vi ser på hvordan ulike ord står i forhold til hverandre.

7.3.1 Figurrekke

Hele kapittel 6 er viet til figurrekker. Disse vil derfor ikke bli videre diskutert her.

7.3.2 Tallrekke

Hele kapittel 5 er viet til tallrekker. Disse vil derfor ikke bli videre diskutert her.

7.3.3 Bokstavrekke

Bokstavrekker er til forveksling like tallrekker. Som vi skal se er det ofte bare å oversette bokstavenes nummer i alfabetet til tall og bruke tallrekkeimplementasjonen fra kapittel 5 til å løse oppgaven.

7.3.3.1 Eksempeloppgave

Oppgave (noe modifisert, hentet fra Séréville og Myers bok om hvordan man får suksess i IQ-tester [dSM94, oppgave 148], samme kilde forteller at man har tretti sekunder på å løse oppgaven):

> Fortsett følgende bokstavrekke:
> A, E, I, M, Q, ?

Som vi ser er oppgaven gitt uten svaralternativer. Dersom vi enkelt og greit oversetter bokstavene til tallet bokstaven har i alfabetet får vi følgende tallrekke: 1, 5, 9, 13, 17. Som vi ser er hvert tall i rekken fire høyere enn det forrige. Neste tall i rekken blir dermed 21, eller U, som er den riktige bokstaven, noe tallrekkeimplementasjonen fra kapittel 5 raskt ville funnet.

7.3.3.2 Hvordan løse lignende oppgaver

Det finnes enkelte varianter av bokstavrekker som ikke alltid gjør dem like enkle. Den verste er naturligvis rekker der bokstavene har en spesiell betydning. Eksempel på en slik rekke kan være J, F, M, A, M, J der hver bokstav tilsvarer første bokstav i månedsnavn fra januar og utover. Slik rekker er nesten identiske med tallrekker vi tidligere har omtalt som "jukserekker", og vil derfor ikke bli behandlet videre annet enn det som står i kapittel 5.2.4.

Det er imidlertid ikke alltid at rekkene tilsvarer samtlige bokstaver i alfabetet. Dersom ikke tallrekkeimplementasjonen klarer å løse oppgaven bør man derfor prøve å lage en

rekke uten enkelte av bokstavene fra alfabetet, enten vokalene eller konsonantene. Man bør også være klar over at alfabetet starter på nytt når det ikke er nok bokstaver, slik at A kommer etter Å (eller Z hvis man bruker det engelsk alfabetet). Det gjør at A kan få verdien 30 (eller 27) i stedet for 1 i enkelte tilfeller.

7.3.4 Matrise med figurer

Oppgaver med figurmatrise ligner mye på oppgaver med figurrekker (kapittel 7.3.1), men de innholder noen ulikheter, og vi vil derfor se på en eksempeloppgave.

7.3.4.1 Eksempeloppgave

Oppgave (hentet fra Test nasjonen 2003, kilden forteller ingenting om hvor lang tid man har på å løse oppgaven [NRK03, oppgave 36]):

> Fullfør den manglende figuren (i figur 7.2)

Den største forskjellen på oppgavene med figurmatrise og figurrekke-oppgavene er at figurene endrer seg i to dimensjoner. Vi får dermed, dersom vi bruker genetisk programmering, to sett med treningseksempler. Det ene av disse vil gjelde for horisontale endringer mens det andre vil gjelde for vertikale endringer. Totalt vil vi ha fem treningseksempler i hver retning.

Bortsett fra dette er oppgaven identisk med figurrekke-oppgavene nevnt tidligere, og mange av idéene fra implementasjonen i kapittel 6 kan derfor brukes.

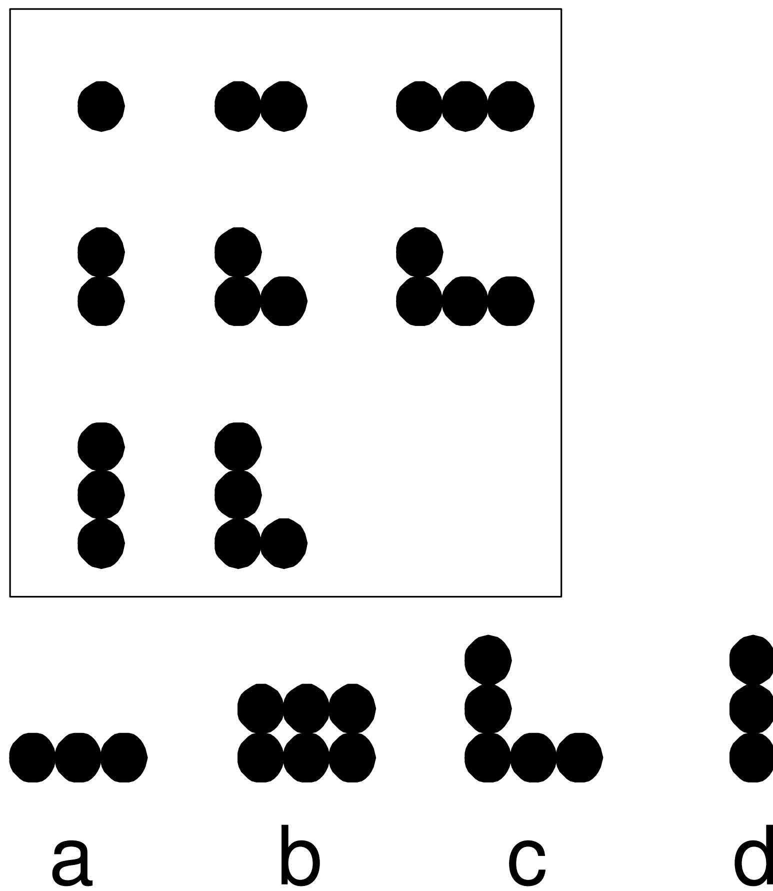
7.3.5 Ordpar

Ordpar er ord som står i forhold til hverandre på en logisk måte, noe eksempeloppgaven i kapittel 7.3.5.1 viser.

7.3.5.1 Eksempeloppgave

Oppgave (hentet fra Test nasjonen 2004, samme kilde forteller at man har tjue sekunder på å løse oppgaven [NRK04c, oppgave 41]):

> Hvilket ord passer?
> SYKEHUS står til LEGE som SKOLE står til?
> a) undervisning
> b) klasse



Figur 7.1: Oppgave med figurmatrise.

c) elev
d) lærer

For å løse oppgaven må vi se på hvilken sammenheng det er mellom ordene sykehus og lege. For eksempel kan vi si at et sykehus er arbeidsplassen til en lege. Dermed vet vi at svaralternativ d er riktig fordi skole er arbeidsplassen til en lærer. Det finnes ingen ordbøker eller oppslagsverk der slike ordpar står. Det finnes likevel en løsningsmåte som vil kunne løse en del slike ordpar.

Verdensveven består av en enorm mengde tekst. Vi kan tenke oss at vi søker etter de to ordene sykehus og lege, og at søket tillater oss å kun ta med treff der det maksimalt er fem andre ord mellom de to søkeordene. Deretter lager vi oss en tabell over alle frasene som finnes mellom de to søkeordene, og grupperer like fraser sammen.

Anta at den vanligste frasen mellom sykehus og lege er nettopp ”er arbeidsplassen til en”. Deretter søker vi på nytt med frasen ”skole er arbeidsplassen til en undervisning” og ser hvor mange treff det gir. Slik søker vi på samme måte for alle svaralternativene og ser hvilket som gir flest treff. Svaralternativet med flest treff antar vi er det riktige svaralternativet.

Det finnes en rekke måter å forbedre forslaget til løsning. Vi bør blant annet ta hensyn til at substantiver av forskjellig kjønn har ulike ubestemte artikler. Sannsynligvis vil vi få flere treff dersom vi oversetter ordene i oppgaven til engelsk (simpelten fordi verdensveven består av mange flere engelske enn norske sider). Løsningsforslaget gitt over med de forbedringer som er mulige er antageligvis en av de beste måtene å løse slike oppgaver på selv om kontakt med verdensveven ikke var et av de hjelpemidlene vi godtok i kapittel 3.2.3...

7.4 Regning

Regneoppgaver gitt i IQ-tester er nesten alltid kamuflert i tekstoppgaver, og slike typer regneoppgaver er derfor de eneste som blir presentert i dette kapitlet. Dersom man ved hjelp av datamaskinen klarer å løse eksempeloppgaven beskrevet i kapittel 7.4.1.1 vil man antageligvis ha en løsning god nok til å løse de fleste andre tekstlig gitte regneoppgaver også.

7.4.1 Tekstoppgaver

Regneoppgaver er, på samme måte som hukommelsesoppgaver, nokså enkle oppgaver for datamaskinen. Regneoppgaver gitt i IQ-tester består hovedsaklig av tekstoppgaver. Utfordringen for et menneske som skal løse slike oppgaver er tredelt: Forstå oppgaven

og hva det spørres om, sette opp et regnestykke på bakgrunn av hva det spørres om, og til slutt regne ut regnestykket.

En datamaskin utfordres på mange måter likt som mennesket i en regneoppgave, men mens det for et menneske kanskje er selve utregningen som utgjør den vanskeligste delen, er det for en datamaskin helt klart forståelse av naturlig språk (forstå oppgaven) og kunnskapsrepresentasjon (representere oppgaven på en måte som gjør det mulig å sette opp regnestykket) som er vanskeligst. Som vi skal se henger disse to utfordringen så sterkt sammen at vi kan regne dem som én stor.

De påfølgende kapitlene inneholder en oppgave fra en reell IQ-test som blir gjennomgått og løst (kapittel 7.4.1.1), og en kort oppsummering av hvordan oppgaver som ligner eksempeloppgaven kan løses (kapittel 7.4.1.2).

7.4.1.1 Eksempeloppgave

Oppgave (hentet fra Test nasjonen 2004, samme kilde forteller at man har tretti sekunder på å løse oppgaven [NRK04c, oppgave 45]):

> 53 kg epler skal fordeles på to kasser. Den største rommer 15 kg mer enn den andre. Hvor mange kg tar den minste?
> a) 24 kg
> b) 16 kg
> c) 21 kg
> d) 19 kg

Vi antar at innskanningen og tekstgjennkjenningen ikke er en nevneverdig utfordring (dersom den skulle være det, se kapittel 3.2.4.1). Det første problemet vi står overfor er dermed naturlig språk-forståelse, et fagfelt som er for stort og komplekst til å bli gjennomgått i detalj her (se for øvrig kapittel 3.2.4.2).

En vanlig måte å representere naturlig språk på i en form som er forståelig for datamaskinen er ved hjelp av predikatlogikk [Nil98, side 425–433]. For eksempeloppgaven over kan dette skrives på følgende måte (ord som begynner med stor bokstav er variabelnavn):

```
kasse1(epler(Kasse1rommer))
kasse2(epler(Kasse2rommer))
like(Kasse1rommer + Kasse2rommer, 53)
like(Kasse1rommer + 15, Kasse2rommer)
svaralt(Svar, Kasse1rommer)

svaralt(a, 24)
svaralt(b, 16)
svaralt(c, 21)
```

```
svaralt(d, 19)
```

Ved å bruke backward chaining [Nil98, side 274–275] kan man da finne ut hvor mye den minste kassen veier. Med backward chaining prøver vi først et av svaralternativene og ser om dette stemmer. Først antar vi at svaralternativ a stemmer, og vi erstatter derfor `Svar`med `a`og `Kasse1rommer med 24`:

```
svaralt(Svar, Kasse1rommer) -> svaralt(a, 24)
like(Kasse1rommer + 15, Kasse2rommer) ->
    like(24 + 15, Kasse2rommer) -> like(24 + 15, 39)
like(Kasse1rommer + Kasse2rommer, 53) ->
    like(24 + 39, 53) -> Feil!
```

Som vi ser fører løsningsalternativ a til feil, og vi prøver derfor på samme måte med svarsalternativ b:

```
svaralt(Svar, Kasse1rommer) -> svaralt(b, 16)
like(Kasse1rommer + 15, Kasse2rommer) ->
    like(16 + 15, Kasse2rommer) -> like(16 + 15, 31)
like(Kasse1rommer + Kasse2rommer, 53) ->
    like(16 + 31, 53) -> Feil!
```

Også svaralternativ b feilet, så vi fortsetter med å prøve med svaralternativ c:

```
svaralt(Svar, Kasse1rommer) -> svaralt(c, 21)
like(Kasse1rommer + 15, Kasse2rommer) ->
    like(21 + 15, Kasse2rommer) -> like(21 + 15, 36)
like(Kasse1rommer + Kasse2rommer, 53) ->
    like(21 + 36, 53) -> Feil!
```

De tre første alternativene feilet, og vi prøver derfor til slutt med svaralternativ d:

```
svaralt(Svar, Kasse1rommer) -> svaralt(d, 19)
like(Kasse1rommer + 15, Kasse2rommer) ->
    like(19 + 15, Kasse2rommer) -> like(19 + 15, 34)
like(Kasse1rommer + Kasse2rommer, 53) ->
    like(19 + 34, 53) -> Riktig!
```

Endelig fant programmet det riktige svaret, nemlig svaralternativ d. Som nevnt er naturlig språk-forståelse et stort og komplekst fagfelt, så bare en liten del av hva som trengs for å løse en slik oppgave er vist her. Dersom man vil prøve å få datamaskinen til å gjøre alle stegene i en slik oppgave anbefales det å lese for eksempel Jurafsky og Martins bok om tale- og språkprosessering [JM00].

7.4.1.2 Hvordan løse lignende oppgaver

Fremgangsmåten for å løse andre tekstlig gitte regneoppgaver skiller seg lite fra eksempelet fra kapittel 7.4.1.1 selv for vanskelige oppgaver. Nøkkelen for å løse oppgaven er, som i eksempelet, å representere oppgaveteksten ved hjelp av predikatlogikk.

Dette er den eneste store utfordringen en datamaskin har når den skal løse tekstlig gitte regneoppgaver.

7.5 Teknikk og romforståelse

Som vi skal se strekker oppgavene i denne kategorien seg over et vidt spekter. Alt fra teknikkoppgaver (kapittel 7.5.1) til speilbilder (kapittel 7.5.2) og utbrettede 3D-figurer (kapittel 7.5.3) finnes her.

7.5.1 Teknikkoppgaver

Teknikkoppgavene gitt i Test nasjonens to IQ-tester [NRK04c, oppgave 51–56] [NRK03, oppgave 53–58] er svært ulike og, sett fra en datamaskins perspektiv, umulige å løse med dagens teknologi. Oppgavene ligner lite eller ingenting på de andre oppgavene gitt i testene. Flere av oppgavene krever i tillegg spesiell kunnskaper for å løses, og er derfor kulturspesifikke. Oppgavetypen vil derfor ikke bli omtalt videre her.

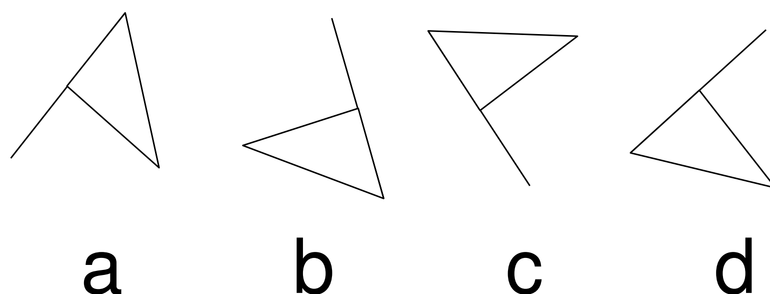
7.5.2 Speilbilder

Speilbildeoppgaver kjennetegnes ved at fire figurer er rotert. Tre av figurene er like (med unntak av rotasjonen), mens den siste figuren er speilbilde av de andre. Oppgaven går ut på å finne ut hvilken figur som er speilbilde. Det har i utgangspunktet ikke noe å si hvordan den figuren som er en speilet versjon av de andre ser ut, da vår oppgave kun går ut på å finne ut hvilke figurer som er like.

7.5.2.1 Eksempeloppgave

Oppgave (hentet fra Test nasjonen 2003, kilden forteller ingenting om hvor lang tid man har på å løse oppgaven [NRK03, oppgave 64]):

> Hvilken figur (i figur 7.2) passer ikke i rekken?



Figur 7.2: Oppgave med speilbilder.

For å løse oppgaven kan vi hente mange prinsipper fra hvordan dette kan gjøres for å løse figurrekker. Prinsippet er beskrevet i kapittel 6.2.2, og forteller hvordan vi kan rotere en figur i forhold til en annen og på den måten sammenligne om figurene har samme form. Vi finner da ut at tre av figurene har lik form. Hva slags form den fjerde og siste figuren har er uten betydning.

7.5.3 Utbrettede 3D-figurer

Oppgavene representert i denne oppgavetypen går ut på å sammenligne 3D-figurer med ulike brettemønstre.

7.5.3.1 Eksempeloppgave

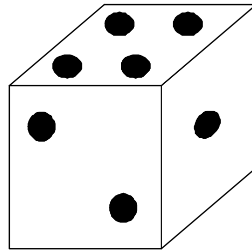
Oppgave (hentet fra Test nasjonen 2003, kilden forteller ingenting om hvor lang tid man har på å løse oppgaven [NRK03, oppgave 70]):

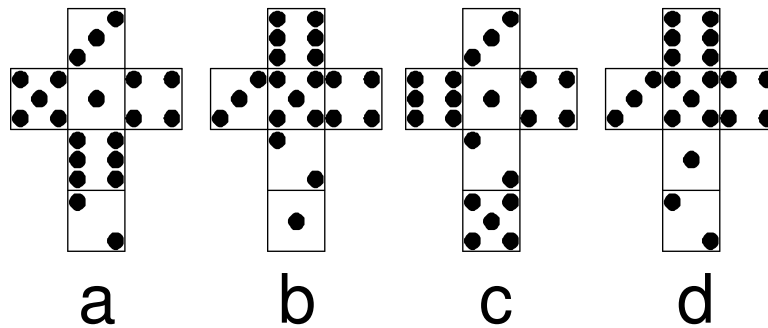
> Hvilket brettemønster blir til figuren som er gitt (se figur 7.3)?

Det finnes mye programvare som konverterer 2D-figurer til 3D-figurer [Inc05] [IC02]. Én mulig løsning vil derfor være å konvertere de utbrettede svaralternativene til 3D-figurer og deretter sammenligne disse med 3D-figuren fra oppgaven. Når figurene er konvertert kan disse sammenlignes ved å rotere 3D-figurene rundt alle akser og på den måten finne ut hvilken av utbrettfigurene som ligner mest på 3D-figuren fra oppgaven.

7.5.3.2 Hvordan løse lignende oppgaver

Vi kan tenke oss oppgaver som er motsatt av eksempeloppgaven, altså oppgaver der brettemønsteret er gitt og vi skal finne hvilken av en rekke 3D-figurer mønsteret tilsvarer. Vi kan også tenke oss oppgaver der brettemønsteret er mye mer komplisert enn en





Figur 7.3: Oppgave med utbrettede 3D-figurer.

terning. Likevel vil prinsippene være så like eksempeloppgaven fra kapittel 7.5.3.1 at de ikke vil bli diskutert videre her.

Kapittel 8

Konklusjon

Gjennom å se på hvordan man kan løse ulike IQ-oppgaver har denne rapporten vist at en datamaskin vil kunne score relativt bra i en IQ-test. Rapporten inneholder en detaljert beskrivelse av alle stegene som trengs for å løse en IQ-test fra begynnelse til slutt. I tillegg inneholder den to implementasjoner for å finne henholdsvis neste tall i en tallrekke og neste figur i en figurrekke, og en beskrivelse av hvordan vi kan løse en rekke andre IQ-oppgaver ved hjelp av datamaskinen.

Rapporten har vist at ulike teknikker må brukes for å løse ulike oppgaver. Spesielt har vi tatt for oss genetisk programmering som er velegnet til å løse ulike oppgaver basert på logikk. De to implementasjonene som begge var basert på genetisk programmering viste utmerkede resultater.

Implementasjonene og beskrivelsene av hvordan andre oppgaver kan løses danner grunnlaget for et estimat av hvor godt en datamaskin vil score på en IQ-test. Estimatet med begrunnelse og diskusjon finnes i vedlegg G.

Estimatet viser at datamaskinen generelt er god på oppgaver som innbefatter logikk eller hukommelse, og mindre god på oppgaver som innbefatter god språkforståelse eller gode tekniske kunnskaper. Vi kunne derfor tenke oss følgende fremskritt for at datamaskinen skal score bedre på IQ-tester enn den gjør i dag:

- Kunnskapsdatabaser for nær sagt alle emner. Spesielt er en kunnskapsdatabase med godt grensesnitt for å løse problemer som innbefatter fysikk svært velkommen.
- Naturlig språk-tolkere som er mer generelle enn dagens.
- Ordbøker som viser relasjoner mellom ord som er bedre enn de som allerede eksisterer.
- Gode oversettere, noe som vil gi den fordelen at ikke IQ-oppgavene må være gitt på samme språk som eventuelle ordbøker eller kunnskapsdatabaser.

Det finnes riktignok løsninger som gjør at alle fire ønskene er innfridd. Men dessverre er de bare delvis innfridd. Systemene som trengs er store og komplekse å utvikle. Å lage slike kun for at datamaskinen skal score godt på IQ-tester virker derfor meningsløst. Ønskene over er derimot svært velkomne av mange brukere til helt andre oppgaver enn å løse IQ-tester, og vil gjøre at en datamaskins kunstige intelligens blir bedre. Gode kunnskapsdatabaser vil for eksempel gjøre at mange flere kan anvende en kunnskap de mangler.[1]

Dersom ønskene relatert til språk blir utviklet vil dette lette kommunikasjonen, både mellom mennesker og mellom datamaskin og menneske. Gode og generelle naturlig språk-tolkere vil gjøre det lettere å finne informasjon ettersom man kan stille spørsmålene slik man ville stilt dem til et menneske, og gode oversettere gjør at man både vil kunne lese tekster skrevet på andre språk, og kunne kommunisere med personer som ikke snakker samme språk som en selv, på en enkel og rask måte.

Det viser seg altså at hjelpemidler som brukes for å løse IQ-tester også er nyttige innenfor en rekke andre områder. Kanskje en datamaskins IQ-test-resultat om noen år kan sammenlignes med en IQ-test utført av en av dagens datamaskiner og dermed brukes til å sammenligne hvor gode menneskene har vært til å utvikle en rekke nyttige programmer og kunnskapsdatabaser?

[1] For eksempel vil en kunnskapsdatabase innen fysikk kunne gi svar på om man kaster lenger med en fiskestang som er tre meter enn en som bare er to meter.

Tillegg A

Kildekode, tallrekker

I dette kapitlet finnes kildekoden til alle klassene som ikke følger ECJ (kapittel 4.3.2). I tillegg er parameterfila som brukes for å kjøre programmet lagt her. Koden er kommentert ved hjelp av Javadoc i tillegg til noen utfyllende kommentarer. For de metodene som ikke har Javadoc, se metodens superklasse.

A.1 Add.java

```
package ec.app.number;

import ec.EvolutionState;
import ec.Problem;
import ec.gp.ADFStack;
import ec.gp.GPData;
import ec.gp.GPIndividual;
import ec.gp.GPNode;
import ec.util.Parameter;

/**
 * Klassen er en funksjon (node) i GP-treet. Funksjonen
 * legger sammen to tall.
 *
 * @author Anders Lund Fredriksen <anderf@stud.ntnu.no>
 */
public class Add extends GPNode {
    public String toString() {
        return "+";
    }
```

```
    public void checkConstraints(final EvolutionState state,
            final int tree,
            final GPIndividual typicalIndividual,
            final Parameter individualBase) {
        super.checkConstraints(state, tree, typicalIndividual,
                individualBase);
        if (children.length != 2)
            state.output.error(
                    "Incorrect number of children for node " +
                    toStringForError() + " at " +
                    individualBase);
    }

    public void eval(final EvolutionState state,
                     final int thread,
                     final GPData input,
                     final ADFStack stack,
                     final GPIndividual individual,
                     final Problem problem) {
        int result;
        NumberData rd = ((NumberData)(input));

        children[0].eval(state, thread, input, stack,
                individual, problem);
        result = rd.x;

        children[1].eval(state, thread, input, stack,
                individual, problem);
        int t = rd.x;
        rd.x = result + t;
    }
}
```

A.2 Div.java

```
package ec.app.number;

import ec.EvolutionState;
import ec.Problem;
import ec.gp.ADFStack;
```

```
import ec.gp.GPData;
import ec.gp.GPIndividual;
import ec.gp.GPNode;
import ec.util.Parameter;

/**
 * Klassen er en funksjon (node) i GP-treet. Funksjonen
 * dividerer to tall.
 *
 * @author Anders Lund Fredriksen <anderf@stud.ntnu.no>
 */
public class Div extends GPNode {
    public String toString() {
        return "/";
    }

    public void checkConstraints(final EvolutionState state,
            final int tree,
            final GPIndividual typicalIndividual,
            final Parameter individualBase) {
        super.checkConstraints(state, tree, typicalIndividual,
                individualBase);
        if (children.length != 2)
            state.output.error(
                    "Incorrect number of children for node " +
                    toStringForError() + " at " +
                    individualBase);
    }

    public void eval(final EvolutionState state,
                     final int thread,
                     final GPData input,
                     final ADFStack stack,
                     final GPIndividual individual,
                     final Problem problem) {
        int result;
        NumberData rd = ((NumberData)(input));

        children[0].eval(state, thread, input, stack,
                individual, problem);
        result = rd.x;

        children[1].eval(state, thread, input, stack,
```

```
                individual, problem);

        if (result == 0) {
            rd.x = 0;
        } else if (rd.x == 0) {
            rd.x = 10000;
        } else {
            rd.x = result / rd.x;
        }
    }
}
```

A.3 I.java

```
package ec.app.number;

import ec.EvolutionState;
import ec.Problem;
import ec.gp.ADFStack;
import ec.gp.GPData;
import ec.gp.GPIndividual;
import ec.gp.GPNode;
import ec.util.Parameter;

/**
 * Klassen er en variabel (node) i GP-treet.
 *
 * @author Anders Lund Fredriksen <anderf@stud.ntnu.no>
 */
public class I extends GPNode {
    private static String name = "I";

    public String toString() {
        return name;
    }

    /**
     * Metoden endrer navn på noden.
     *
     * @param n Nodens nye navn.
     */
```

```
    public static void setName(String n) {
        name = n;
    }

    public void checkConstraints(final EvolutionState state,
            final int tree,
            final GPIndividual typicalIndividual,
            final Parameter individualBase) {
        super.checkConstraints(state,tree,typicalIndividual,
                individualBase);
        if (children.length != 0)
            state.output.error(
                    "Incorrect number of children for node " +
                    toStringForError() + " at " +
                    individualBase);
    }

    public void eval(final EvolutionState state,
                     final int thread,
                     final GPData input,
                     final ADFStack stack,
                     final GPIndividual individual,
                     final Problem problem) {
        NumberData rd = ((NumberData)(input));
        rd.x = ((Number)problem).currentI;
    }
}
```

A.4 Mul.java

```
package ec.app.number;

import ec.EvolutionState;
import ec.Problem;
import ec.gp.ADFStack;
import ec.gp.GPData;
import ec.gp.GPIndividual;
import ec.gp.GPNode;
import ec.util.Parameter;

/**
```

```
 * Klassen er en funksjon (node) i GP-treet. Funksjonen
 * multipliserer to tall.
 *
 * @author Anders Lund Fredriksen <anderf@stud.ntnu.no>
 */
public class Mul extends GPNode {
    public String toString() {
        return "*";
    }

    public void checkConstraints(final EvolutionState state,
            final int tree,
            final GPIndividual typicalIndividual,
            final Parameter individualBase) {
        super.checkConstraints(state, tree, typicalIndividual,
                individualBase);
        if (children.length != 2)
            state.output.error(
                    "Incorrect number of children for node " +
                    toStringForError() + " at " +
                    individualBase);
    }

    public void eval(final EvolutionState state,
                     final int thread,
                     final GPData input,
                     final ADFStack stack,
                     final GPIndividual individual,
                     final Problem problem) {
        int result;
        NumberData rd = ((NumberData)(input));

        children[0].eval(state, thread, input, stack,
                individual, problem);
        result = rd.x;

        children[1].eval(state, thread, input, stack,
                individual, problem);
        int t = rd.x;
        rd.x = result * t;
    }
}
```

A.5 Number.java

```
package ec.app.number;

import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;

import ec.EvolutionState;
import ec.Fitness;
import ec.Individual;
import ec.gp.GPIndividual;
import ec.gp.GPProblem;
import ec.gp.koza.KozaFitness;
import ec.simple.SimpleProblemForm;
import ec.util.Parameter;

/**
 * Klassen leser inn rekke og svaralternativer fra tastaturet,
 * og evaluerer seg frem til mulig løsning.
 *
 * Se kapittel 5 i rapporten for grundigere beskrivelse av
 * implementasjonen.
 *
 * @author Anders Lund Fredriksen <anderf@stud.ntnu.no>
 */
public class Number extends GPProblem implements
SimpleProblemForm {
    public static final String P_DATA = "data";
    public int currentX1, currentX2, currentI;
    public NumberData input;
    public int[] x = new int[6]; //Tallrekken
    private int sola; //Svaralternativ a
    private int solb; //Svaralternativ b
    private int solc; //Svaralternativ c
    private int sold; //Svaralternativ d
    public static boolean tested = false;
    private static boolean x1IsOne = false;
    private static boolean x2IsOne = false;
    private static int gen = 5;
    private final BufferedReader in =
        new BufferedReader(new InputStreamReader(System.in));
```

```
    public Object protoClone() throws
     CloneNotSupportedException {
        Number newobj = (Number)(super.protoClone());
        newobj.input = (NumberData)(input.protoClone());
        return newobj;
    }

    public void setup(final EvolutionState state,
            final Parameter base) {
        super.setup(state,base);

        //Setter opp input. Uttrygt å bruke standard.
        input = (NumberData)
         state.parameters.getInstanceForParameterEq(
            base.push(P_DATA), null, NumberData.class);
        input.setup(state,base.push(P_DATA));

        //Leser inn rekke og svaralternativer fra tastaturet.
        String inline = "";

        System.out.print("Skriv inn tallrekke (6 heltall," +
         "adskilt av mellomrom, avsluttet med enter): ");
        try {
            inline = in.readLine();
        } catch (IOException e) { }
        if (inline != null) {
            String[] splitted = inline.split(" ");
            try {
                for (int i = 0; i < splitted.length; i++) {
                    x[i] = Integer.parseInt(splitted[i]);
                }
            } catch (NumberFormatException e) { }
        }

        System.out.print("Skriv inn svaralternativ a " +
         "(heltall, avsluttet med enter): ");
        try {
            inline = in.readLine();
        } catch (IOException e) { }
        if (inline != null) {
            try {
                sola = Integer.parseInt(inline);
```

```
            } catch (NumberFormatException e) { }
        }

        System.out.print("Skriv inn svaralternativ b " +
         "(heltall, avsluttet med enter): ");
        try {
            inline = in.readLine();
        } catch (IOException e) { }
        if (inline != null) {
            try {
                solb = Integer.parseInt(inline);
            } catch (NumberFormatException e) { }
        }

        System.out.print("Skriv inn svaralternativ c " +
         "(heltall, avsluttet med enter): ");
        try {
            inline = in.readLine();
        } catch (IOException e) { }
        if (inline != null) {
            try {
                solc = Integer.parseInt(inline);
            } catch (NumberFormatException e) { }
        }

        System.out.print("Skriv inn svaralternativ d " +
         "(heltall, avsluttet med enter): ");
        try {
            inline = in.readLine();
        } catch (IOException e) { }
        if (inline != null) {
            try {
                sold = Integer.parseInt(inline);
            } catch (NumberFormatException e) { }
        }
    }

    public void evaluate(EvolutionState state, Individual ind,
            int threadnum) {
        if (state.generation < 5) {
            evaluate1(state, ind, threadnum);
        } else if (state.generation < 20) {
            if (state.generation == gen) {
```

```
        try {
            for (int i = 0;i < state.population.
            subpops[0].individuals.length; i++) {
            state.population.subpops[0].
            individuals[i] =
                state.population.subpops[0].
                individuals[i].species.
                newIndividual(
                state,state.population.subpops[0],
                (Fitness)
                (state.population.subpops[0].
                        f_prototype.protoClone()));

            KozaFitness f = ((KozaFitness)state.
                    population.subpops[0].
                    individuals[i].fitness);
            f.setFitness(state, (float)1000);
            f.hits = 0;
            state.population.subpops[0].
            individuals[i].evaluated = true;
            }
            gen = 20;
        } catch (CloneNotSupportedException e) {
            e.printStackTrace();
        }
        }
        evaluate3(state, ind, threadnum);
    } else if (state.generation < 25) {
        if (state.generation == gen) {
        try {
            for (int i = 0; i < state.population.
            subpops[0].individuals.length; i++) {
            state.population.subpops[0].
            individuals[i] =
                state.population.subpops[0].
                individuals[i].species.
                newIndividual(
                state,state.population.subpops[0],
                (Fitness)
                (state.population.subpops[0].
                        f_prototype.protoClone()));

            KozaFitness f = ((KozaFitness)state.
```

```
                        population.subpops[0].
                        individuals[i].fitness);
                f.setFitness(state, (float)1000);
                f.hits = 0;
                state.population.subpops[0].
                individuals[i].evaluated = true;
                }
                gen = 100;
            } catch (CloneNotSupportedException e) {
                e.printStackTrace();
            }
            }
            evaluate2(state, ind, threadnum);
        } else {
            evaluate3(state, ind, threadnum);
        }
    }

    /**
     * Lineær rekke.
     *
     * @param state Tilstand.
     * @param ind Individ.
     * @param threadnum Trådnummer
     */
    public void evaluate1(EvolutionState state, Individual ind,
            int threadnum) {
        if (!ind.evaluated) {  //Ikke evaluer på nytt.
            X1.setName("ONE");
            X2.setName("X");
            I.setName("I");

            x1IsOne = true;
            x2IsOne = false;

            currentX1 = 1;

            int hits = 0;
            int sum = 0;
            int expectedResult;
            int result = 0;

            for (int i = 0; i < x.length - 1; i++) {
```

```
                currentX2 = x[i];
                currentI = i+1;
                expectedResult = x[i+1];

                ((GPIndividual)ind).trees[0].child.eval(
                        state,threadnum,input,stack,
                        ((GPIndividual)ind),this);

                if (Math.abs(expectedResult - input.x) == 0) {
                    hits++;
                }

                result += Math.abs(expectedResult - input.x);
            }

            sum = result;

            if (sum < 0) {
                sum = 10000;
            }

            //Vi bruker Koza-fitness.
            KozaFitness f = ((KozaFitness)ind.fitness);
            f.setFitness(state, (float)sum);
            f.hits = hits;
            ind.evaluated = true;
        }
    }

    /**
     * Alternerende rekke.
     *
     * @param state Tilstand.
     * @param ind Individ.
     * @param threadnum Trådnummer
     */
    public void evaluate2(EvolutionState state, Individual ind,
            int threadnum) {
        if (!ind.evaluated) { //Ikke evaluer på nytt.
            X1.setName("ONE");
            X2.setName("X");
            I.setName("I");
```

```
currentX1 = 1;
x1IsOne = true;
x2IsOne = false;

int hits = 0;
int sum = 0;
int expectedResult;
int result = 0;

currentX2 = x[1];
currentI = 2;
expectedResult = x[2];

((GPIndividual)ind).trees[0].child.eval(
        state,threadnum,input,stack,
        ((GPIndividual)ind),this);

if (Math.abs(expectedResult - input.x) == 0) {
    hits++;
}

result += Math.abs(expectedResult - input.x);

//--------------------------------------

currentX2 = x[3];
currentI = 4;
expectedResult = x[4];

((GPIndividual)ind).trees[0].child.eval(
        state,threadnum,input,stack,
        ((GPIndividual)ind),this);

if (Math.abs(expectedResult - input.x) == 0) {
    hits++;
}

result += Math.abs(expectedResult - input.x);

sum = result;

if (sum < 0) {
    sum = 10000;
```

```
            }

            //Vi bruker Koza-fitness.
            KozaFitness f = ((KozaFitness)ind.fitness);
            f.setFitness(state, (float)sum);
            f.hits = hits;
            ind.evaluated = true;
        }
    }

    /**
     * Flervariabelrekke.
     *
     * @param state Tilstand.
     * @param ind Individ.
     * @param threadnum Trådnummer
     */
    public void evaluate3(EvolutionState state, Individual ind,
            int threadnum) {
        if (!ind.evaluated) { //Ikke evaluer på nytt.
            X1.setName("X1");
            X2.setName("X2");
            I.setName("I");
            x1IsOne = false;
            x2IsOne = false;

            int hits = 0;
            int sum = 0;
            int expectedResult;
            int result = 0;

            for (int i = 0; i < x.length - 2; i++) {
                currentX1 = x[i];
                currentX2 = x[i+1];
                currentI = i+2;
                expectedResult = x[i+2];

                ((GPIndividual)ind).trees[0].child.eval(
                        state,threadnum,input,stack,
                        ((GPIndividual)ind),this);

                if (Math.abs(expectedResult - input.x) == 0) {
                    hits++;
```

```
                }

                result += Math.abs(expectedResult - input.x);
            }

            sum = result;

            if (sum < 0) {
                sum = 10000;
            }

            //Vi bruker Koza-fitness.
            KozaFitness f = ((KozaFitness)ind.fitness);
            f.setFitness(state, (float)sum);
            f.hits = hits;
            ind.evaluated = true;
        }
    }

    public void describe(Individual ind,
            EvolutionState state,
            int threadnum,
            int log,
            int verbosity) {
        System.out.println("");
        System.out.println("=========RESULTAT==========");

        if (x1IsOne) {
            currentX1 = 1;
        } else {
            currentX1 = x[x.length - 2];
        }

        if (x2IsOne) {
            currentX2 = 1;
        } else {
            currentX2 = x[x.length - 1];
        }

        currentI = x.length;

        ((GPIndividual)ind).trees[0].child.eval(
                state,threadnum,input,stack,
```

```
                ((GPIndividual)ind),this);
        System.out.println("Svar (beregnet): " + input.x);

        //Sjekk hvilket svaralternativ som er nærmest
        System.out.print("Svaralternativ: ");
        if (Math.abs(input.x - sola) <=
            Math.abs(input.x - solb) &&
                Math.abs(input.x - sola) <=
                    Math.abs(input.x - solc) &&
                Math.abs(input.x - sola) <=
                    Math.abs(input.x - sold)) {
            System.out.println("a");
        } else if (Math.abs(input.x - solb) <=
            Math.abs(input.x - sola) &&
                Math.abs(input.x - solb) <=
                    Math.abs(input.x - solc) &&
                Math.abs(input.x - solb) <=
                    Math.abs(input.x - sold)) {
            System.out.println("b");
        } else if (Math.abs(input.x - solc) <=
            Math.abs(input.x - sola) &&
                Math.abs(input.x - solc) <=
                    Math.abs(input.x - solb) &&
                Math.abs(input.x - solc) <=
                    Math.abs(input.x - sold)) {
            System.out.println("c");
        } else if (Math.abs(input.x - sold) <=
            Math.abs(input.x - sola) &&
                Math.abs(input.x - sold) <=
                    Math.abs(input.x - solb) &&
                Math.abs(input.x - sold) <=
                    Math.abs(input.x - solc)) {
            System.out.println("d");
        }
    }
}
```

A.6 NumberData.java

```
package ec.app.number;
```

```
import ec.gp.GPData;

/**
 * Klassen tar var på nummerdata og tilbyr en metode for å
 * kopiere disse.
 *
 * @author Anders Lund Fredriksen <anderf@stud.ntnu.no>
 */
public class NumberData extends GPData {
    public int x;

    public GPData copyTo(final GPData gpd) {
        ((NumberData)gpd).x = x;
        return gpd;
    }
}
```

A.7 NumberStatistics.java

```
package ec.app.number;

import ec.EvolutionState;
import ec.gp.koza.KozaStatistics;
import ec.simple.SimpleProblemForm;
import ec.util.Output;

/**
 * Klassen gjør at vi kan skrive ut statistikk for kjøring
 * på den måten vi selv ønsker.
 *
 * @author Anders Lund Fredriksen <anderf@stud.ntnu.no>
 */
public class NumberStatistics extends KozaStatistics {
    public NumberData input;

    public void finalStatistics(final EvolutionState state,
            final int result) {
        super.finalStatistics(state, result);

        ((SimpleProblemForm)(
                state.evaluator.p_problem.protoCloneSimple())).
```

```
                describe(best_of_run[0], state, 0,
                        statisticslog, Output.V_NO_GENERAL);
    }
}
```

A.8 One.java

```
package ec.app.number;

import ec.EvolutionState;
import ec.Problem;
import ec.gp.ADFStack;
import ec.gp.GPData;
import ec.gp.GPIndividual;
import ec.gp.GPNode;
import ec.util.Parameter;

/**
 * Klassen er en konstant (node) i GP-treet.
 *
 * @author Anders Lund Fredriksen <anderf@stud.ntnu.no>
 */
public class One extends GPNode {
    public String toString() {
        return "ONE";
    }

    public void checkConstraints(final EvolutionState state,
            final int tree,
            final GPIndividual typicalIndividual,
            final Parameter individualBase) {
        super.checkConstraints(state, tree, typicalIndividual,
                individualBase);
        if (children.length != 0)
            state.output.error(
                    "Incorrect number of children for node " +
                    toStringForError() + " at " +
                    individualBase);
        }

    public void eval(final EvolutionState state,
```

```
                     final int thread,
                     final GPData input,
                     final ADFStack stack,
                     final GPIndividual individual,
                     final Problem problem) {
        NumberData rd = ((NumberData)(input));
        rd.x = 1;
    }
}
```

A.9 Sub.java

```
package ec.app.number;

import ec.EvolutionState;
import ec.Problem;
import ec.gp.ADFStack;
import ec.gp.GPData;
import ec.gp.GPIndividual;
import ec.gp.GPNode;
import ec.util.Parameter;

/**
 * Klassen er en funksjon (node) i GP-treet. Funksjonen
 * trekker to tall fra hverandre.
 *
 * @author Anders Lund Fredriksen <anderf@stud.ntnu.no>
 */
public class Sub extends GPNode {
    public String toString() {
        return "-";
    }

    public void checkConstraints(final EvolutionState state,
            final int tree,
            final GPIndividual typicalIndividual,
            final Parameter individualBase) {
        super.checkConstraints(state, tree, typicalIndividual,
                individualBase);
        if (children.length != 2)
            state.output.error(
```

```
                    "Incorrect number of children for node " +
                    toStringForError() + " at " +
                    individualBase);
    }

    public void eval(final EvolutionState state,
                     final int thread,
                     final GPData input,
                     final ADFStack stack,
                     final GPIndividual individual,
                     final Problem problem) {
        int result;
        NumberData rd = ((NumberData)(input));

        children[0].eval(state, thread, input, stack,
                individual, problem);
        result = rd.x;

        children[1].eval(state, thread, input, stack,
                individual, problem);
        int t = rd.x;
        rd.x = result - t;
    }
}
```

A.10 X1.java

```
package ec.app.number;

import ec.EvolutionState;
import ec.Problem;
import ec.gp.ADFStack;
import ec.gp.GPData;
import ec.gp.GPIndividual;
import ec.gp.GPNode;
import ec.util.Parameter;

/**
 * Klassen er en variabel (node) i GP-treet.
 *
 * @author Anders Lund Fredriksen <anderf@stud.ntnu.no>
```

```
 */
public class X1 extends GPNode {
    private static String name = "X1";

    public String toString() {
        return name;
    }

    /**
     * Metoden endrer navn på noden.
     *
     * @param n Nodens nye navn.
     */
    public static void setName(String n) {
        name = n;
    }

    public void checkConstraints(final EvolutionState state,
            final int tree,
            final GPIndividual typicalIndividual,
            final Parameter individualBase) {
        super.checkConstraints(state, tree, typicalIndividual,
                individualBase);
        if (children.length != 0)
            state.output.error(
                    "Incorrect number of children for node " +
                    toStringForError() + " at " +
                    individualBase);
    }

    public void eval(final EvolutionState state,
                     final int thread,
                     final GPData input,
                     final ADFStack stack,
                     final GPIndividual individual,
                     final Problem problem) {
        NumberData rd = ((NumberData)(input));
        rd.x = ((Number)problem).currentX1;
    }
}
```

A.11 X2.java

```
package ec.app.number;

import ec.EvolutionState;
import ec.Problem;
import ec.gp.ADFStack;
import ec.gp.GPData;
import ec.gp.GPIndividual;
import ec.gp.GPNode;
import ec.util.Parameter;

/**
 * Klassen er en variabel (node) i GP-treet.
 *
 * @author Anders Lund Fredriksen <anderf@stud.ntnu.no>
 */
public class X2 extends GPNode {
    private static String name = "X2";

    public String toString() {
        return name;
    }

    /**
     * Metoden endrer navn på noden.
     *
     * @param n Nodens nye navn.
     */
    public static void setName(String n) {
        name = n;
    }

    public void checkConstraints(final EvolutionState state,
            final int tree,
            final GPIndividual typicalIndividual,
            final Parameter individualBase) {
        super.checkConstraints(state, tree, typicalIndividual,
                individualBase);
        if (children.length != 0)
            state.output.error(
                    "Incorrect number of children for node " +
```

```
                    toStringForError() + " at " +
                    individualBase);
    }

    public void eval(final EvolutionState state,
                     final int thread,
                     final GPData input,
                     final ADFStack stack,
                     final GPIndividual individual,
                     final Problem problem) {
        NumberData rd = ((NumberData)(input));
        rd.x = ((Number)problem).currentX2;
    }
}
```

A.12 number.params

```
parent.0 = ../../gp/koza/koza.params

# Vi har ett funksjonssett av klasse GPFunctionSet.
gp.fs.size = 1
gp.fs.0 = ec.gp.GPFunctionSet

# Vi kaller funksjonssettet "f0".
# Det bruker standard GPFuncInfo-klasse.
gp.fs.0.name = f0
gp.fs.0.info = ec.gp.GPFuncInfo

# Vi har åtte funksjoner i funksjonssettet. Disse er:
gp.fs.0.size = 8
gp.fs.0.func.0 = ec.app.number.X1
gp.fs.0.func.0.nc = nc0
gp.fs.0.func.1 = ec.app.number.One
gp.fs.0.func.1.nc = nc0
gp.fs.0.func.2 = ec.app.number.Add
gp.fs.0.func.2.nc = nc2
gp.fs.0.func.3 = ec.app.number.Sub
gp.fs.0.func.3.nc = nc2
gp.fs.0.func.4 = ec.app.number.Mul
gp.fs.0.func.4.nc = nc2
gp.fs.0.func.5 = ec.app.number.Div
```

```
gp.fs.0.func.5.nc = nc2
gp.fs.0.func.6 = ec.app.number.X2
gp.fs.0.func.6.nc = nc0
gp.fs.0.func.7 = ec.app.number.I
gp.fs.0.func.7.nc = nc0

eval.problem = ec.app.number.Number
eval.problem.data = ec.app.number.NumberData

# Følgende skal nesten alltid være samme som
# eval.problem.data
# Det definerer dataobjektet brukt internt i
# ADF-stakk-konteksten
eval.problem.stack.context.data = ec.app.number.NumberData

breed.elite.0 = 3
generations = 100
seed.0=time
stat = ec.app.number.NumberStatistics
```

Tillegg B

Kjøreeksempel, tallrekker

Under følger et kjøreeksempel der en av oppgavene fra Test nasjonen 2004 [NRK04c, oppgave 31] er brukt som eksempel. Se readme.txt i rotkatalogen på den vedlagte CD-en for beskrivelse av hvordan man kjører programmet.

```
| ECJ
| An evolutionary computation system (version 12)
| Copyright 2004 by Sean Luke
| Contributors: L. Panait, G. Balan, Z. Skolicki, J. Bassett,
|               R. Hubley, and A. Chircop
| URL: http://cs.gmu.edu/~eclab/projects/ecj/
| Mail: sean@cs.gmu.edu
| Date: September 21, 2004
| Current Java: 1.5.0_02 / Java HotSpot(TM)
    Client VM-1.5.0_02-b09
| Required Minimum Java: 1.2


Threads:  breed/1 eval/1
Seed: 2055397413
Setting up
Processing GP Types
Processing GP Node Constraints
Processing GP Function Sets
Processing GP Tree Constraints
Skriv inn tallrekke (6 heltall, adskilt av mellomrom, avsluttet
med enter): 4 16 8 32 16 64
Skriv inn svaralternativ a (heltall, avsluttet med enter): 48
Skriv inn svaralternativ b (heltall, avsluttet med enter): 86
Skriv inn svaralternativ c (heltall, avsluttet med enter): 52
```

```
Skriv inn svaralternativ d (heltall, avsluttet med enter): 32
Initializing Generation 0
Generation 1
Generation 2
Generation 3
Generation 4
Generation 5
Generation 6
Found Ideal Individual

=========RESULTAT==========
Svar (beregnet): 32
Svaralternativ: d
```

Tillegg C

Resultater, tallrekker

Dette vedlegget inneholder, i tabell C.1 og tabell C.2, en oversikt over resultatene fra kjøring av programmet for å løse tallrekker fra kapittel 5. Resultatene fra kjøringen tar ikke hensyn til hvilke svaralternativer som ble presentert sammen med rekken. Programmet har kun tatt rekken og prøvd å finne hvilket tall som er det neste. Testrekkene her hentet fra Test nasjonen 2004 [NRK04c], Séréville og Myers bok om hvordan man får suksess i IQ-tester [dSM94], NRK.no [NRK04b] i tillegg til noen selvlagde. I de tilfellene der kjøringen gav feil resultat er dette merket med **fet** i tabellen.

Videre diskusjon av resultatene finnes i kapittel 5.4.

Tabell C.1: Resultater, tallrekker, del 1.

	Rekke	**Hentet fra**	**Løsning**	**"Formel"**	**Kjøring 1** Utdata/# gen.	**Kjøring 2** Utdata/# gen.	**Kjøring 3** Utdata/# gen.
1	5, 7, 11, 17, 25, 35	NRK (Test nasjonen 2004)	47	+2, +4, +6, +8, osv.	47/1	47/2	47/1
2	5, 8, 6, 9, 7, 10	NRK (Test nasjonen 2004)	8	+3, -2, +3, -2, osv.	8/7	8/7	8/7
3	9, 14, 28, 33, 66, 71	NRK (Test nasjonen 2004)	142	+5, *2, +5, *2, osv.	142/22	142/22	142/22
4	4, 16, 8, 32, 16, 64	NRK (Test nasjonen 2004)	32	*4, /2, *4, /2, osv.	32/7	32/7	32/7
5	14, 6, 18, 11, 33, 27	NRK (Test nasjonen 2004)	81	-8, *3, -7, *3, osv.	**67**/20	81/24	81/23
6	7, 11, 8, 12, 9, 13	NRK (Test nasjonen 2004)	10	+4, -3, +4, -3, osv.	10/7	10/7	10/7
7	4, 5, 7, 10, 14, 19	Séréville og Myers bok	25	+1, +2, +3, +4, osv.	25/1	25/1	25/1
8	9, 10, 12, 13, 15, 16	Séréville og Myers bok	18	+1, +2, +1, +2, osv.	18/11	18/22	18/15
9	40, 37, 35, 32, 30, 27	Séréville og Myers bok	47	-3, -2, -3, -2, osv.	25/24	25/22	25/22
10	3, 6, 12, 24, 48, 96	Séréville og Myers bok	192	*2, *2, *2, *2, osv.	192/1	192/1	192/1
11	13, 14, 28, 29, 58, 59	Séréville og Myers bok	118	+1, *2, +1, *2, osv.	118/12	118/12	118/18
12	4, 12, 11, 33, 32, 96	Séréville og Myers bok	95	*3, -1, *3, -1, osv.	95/19	95/22	95/18
13	4, 11, 32, 95, 284, 851	Séréville og Myers bok	2552	*3-1, *3-1, *3-1, *3-1, osv.	2552/1	2552/2	2552/2
14	128, 64, 32, 16, 8, 4	Séréville og Myers bok	2	/2, /2, /2, /2, osv.	2/1	2/2	2/1
15	57, 66, 58, 65, 59, 64	Séréville og Myers bok	60	+9, -8, +7, -6, osv.	60/33	60/12	60/15
16	2, 3, 5, 8, 13, 21	Selvlaget	34	x3 = x2 + x1	34/7	34/7	34/7
17	1, 2, 2, 4, 8, 32	Selvlaget	256	x3 = x2 * x1	256/7	256/7	256/7
18	3, 5, 8, 13, 22, 39	Selvlaget	72	*2-1, *2-2, *2-3, osv.	72/2	72/2	72/1
19	1, 1, 2, 6, 24, 120	Selvlaget	720	*1, *2, *3, *4, osv.	720/1	720/1	720/1
20	5, 9, 17, 33, 65, 129	Selvlaget	257	*2-1, *2-1, *2-1, osv.	257/1	257/2	257/1
21	30, 29, 27, 26, 24, 23	NRK.no	21	-1, -2, -1, -2, osv.	21/11	21/22	21/23
22	11, 13, 12, 14, 13, 15	NRK.no	14	+2, -1, +2, -1, osv.	14/7	14/7	14/7

Tabell C.2: Resultater, tallrekker, del 2.

Kjøring 4 Utdata/# gen.	**Kjøring 5** Utdata/# gen.	**Kjøring 6** Utdata/# gen.	**Kjøring 7** Utdata/# gen.	**Kjøring 8** Utdata/# gen.	**Kjøring 9** Utdata/# gen.	**Kjøring 10** Utdata/# gen.	**% ganger med riktig resultat**	**Gjennomsnittlig antall gen.**
47/1	47/1	47/1	47/2	47/2	47/1	47/1	100	1,3
8/7	8/7	8/7	8/7	8/7	8/7	8/7	100	7,0
142/22	142/22	142/22	142/22	142/22	142/22	142/22	100	22,0
32/7	32/7	32/7	32/7	32/7	32/7	32/7	100	7,0
81/23	81/23	81/23	81/23	81/23	**56**/17	81/23	80	22,2
10/7	10/7	10/7	10/7	10/7	10/7	10/7	100	7,0
25/1	25/1	25/1	25/1	25/1	25/1	25/1	100	1,0
18/10	18/22	18/22	18/10	18/22	18/22	18/22	100	17,8
25/14	25/23	25/22	25/22	25/22	25/22	25/22	100	21,5
192/1	192/1	192/1	192/1	192/1	192/1	192/1	100	1
118/22	118/22	118/22	118/15	118/9	118/22	118/22	100	17,6
95/22	95/22	95/22	95/13	95/17	95/22	95/22	100	19,9
2552/2	2552/1	2552/2	2552/3	2552/1	2552/2	2552/2	100	1,8
2/2	2/1	2/1	2/1	2/1	2/2	2/1	100	1,3
58/14	60/14	60/100	60/35	**59**/10	60/12	60/12	80	25,7
34/7	34/7	34/7	34/7	34/7	34/7	34/7	100	7,0
256/7	256/7	256/7	256/7	256/7	256/7	256/7	100	7,0
72/1	72/1	72/2	72/1	72/1	72/2	72/1	100	1,4
720/1	720/1	720/1	720/1	720/1	720/1	720/1	100	1,0
257/1	257/2	257/1	257/1	257/1	257/1	257/2	100	1,3
21/25	21/22	21/22	21/22	21/9	21/10	21/9	100	17,5
14/7	14/7	14/7	14/7	14/7	14/7	14/7	100	7,0

Tillegg D

Kildekode, figurrekker

I dette kapitlet finnes kildekoden til alle klassene som ikke følger ECJ (kapittel 4.3.2) for figurrekkeimplementasjonen. I tillegg er parameterfila som brukes for å kjøre programmet lagt her. Koden er kommentert ved hjelp av Javadoc i tillegg til noen utfyllende kommentarer. For de metodene som ikke har Javadoc, se metodens superklasse.

For grundigere beskrivelse av det implementerte systemet, se kapittel 6.3.

D.1 Add.java

```
package ec.app.figure;

import ec.EvolutionState;
import ec.Problem;
import ec.gp.ADFStack;
import ec.gp.GPData;
import ec.gp.GPIndividual;
import ec.gp.GPNode;
import ec.util.Parameter;

/**
 * Klassen er en funksjon (node) i GP-treet. Funksjonen
 * legger sammen to tall.
 *
 * @author Anders Lund Fredriksen <anderf@stud.ntnu.no>
 */
public class Add extends GPNode {
    public String toString() {
```

```
        return "+";
    }

    public void checkConstraints(final EvolutionState state,
            final int tree,
            final GPIndividual typicalIndividual,
            final Parameter individualBase) {
        super.checkConstraints(state, tree, typicalIndividual,
                individualBase);
        if (children.length != 2)
            state.output.error(
                    "Incorrect number of children for node " +
                    toStringForError() + " at " +
                    individualBase);
    }

    public void eval(final EvolutionState state,
                     final int thread,
                     final GPData input,
                     final ADFStack stack,
                     final GPIndividual individual,
                     final Problem problem) {
        int result;
        FigureData rd = ((FigureData)(input));

        children[0].eval(state, thread, input, stack,
                individual, problem);
        result = rd.x;

        children[1].eval(state, thread, input, stack,
                individual, problem);
        int t = rd.x;
        rd.x = result + t;

    }
}
```

D.2 FigureData.java

```
package ec.app.figure;
```

```
import ec.gp.GPData;

/**
 * Klassen tar var på figurdata og tilbyr en metode for å
 * kopiere disse.
 *
 * @author Anders Lund Fredriksen <anderf@stud.ntnu.no>
 */
public class FigureData extends GPData {
    public int x;

    public GPData copyTo(final GPData gpd) {
        ((FigureData)gpd).x = x;
        return gpd;
    }
}
```

D.3 Figures.java

```
package ec.app.figure;

import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;

import ec.EvolutionState;
import ec.Fitness;
import ec.Individual;
import ec.gp.GPIndividual;
import ec.gp.GPProblem;
import ec.gp.koza.KozaFitness;
import ec.simple.SimpleProblemForm;
import ec.util.Parameter;

/**
 * Klassen leser inn rekke og svaralternativer fra tastaturet,
 * og evaluerer seg frem til mulig løsning.
 *
 * Se kapittel 6 i rapporten for grundigere beskrivelse av
 * implementasjonen.
 *
```

```
 * @author Anders Lund Fredriksen <anderf@stud.ntnu.no>
 */
public class Figures extends GPProblem implements
SimpleProblemForm {
    public static final String P_DATA = "data";
    public int currentX;
    public FigureData input;

    //Figurrekke som skal løses
    private static int x[] = new int[3];
    private static int y[] = new int[3];
    private static int width[] = new int[3];
    private static int height[] = new int[3];
    private static int figrotation[] = new int[3];
    private static int middledist[] = new int[3];
    private static int color[] = new int[3];
    private static int objrotation[] = new int[3];

    //Svaralternativer som skal sjekkes
    private static int losx[] = new int[4];
    private static int losy[] = new int[4];
    private static int loswidth[] = new int[4];
    private static int losheight[] = new int[4];
    private static int losfigrotation[] = new int[4];
    private static int losmiddledist[] = new int[4];
    private static int loscolor[] = new int[4];
    private static int losobjrotation[] = new int[4];

    //Parametrene evolveringen finner
    private static int solx;
    private static int soly;
    private static int solwidth;
    private static int solheight;
    private static int solfigrotation;
    private static int solmiddeldist;
    private static int solcolor;
    private static int solobjrotation;

    //Setter om en parameter er funnet eller ikke
    private static boolean xfound = false;
    private static boolean yfound = false;
    private static boolean widthfound = false;
    private static boolean heightfound = false;
```

```
    private static boolean figrotationfound = false;
    private static boolean middeldistfound = false;
    private static boolean colorfound = false;
    private static boolean objrotationfound = false;

    private static int step = 1;
    private static int gen = 0;
    private static int genbase = 5000;
    private static int expectedAltA = 0;
    private static int expectedAltB = 0;
    private static int expectedAltC = 0;
    private static int expectedAltD = 0;

    private final BufferedReader in =
        new BufferedReader(new InputStreamReader(System.in));

    public Object protoClone() throws
     CloneNotSupportedException {
        Figures newobj = (Figures)(super.protoClone());
        newobj.input = (FigureData)(input.protoClone());
        return newobj;
    }

    public void setup(final EvolutionState state,
            final Parameter base) {
        super.setup(state,base);

        //Setter opp input. Uttrygt å bruke standard.
        input = (FigureData)
         state.parameters.getInstanceForParameterEq(
            base.push(P_DATA), null, FigureData.class);
        input.setup(state,base.push(P_DATA));

        //Leser inn rekke og svaralternativer fra tastaturet.
        String inline = "";

        System.out.println("");
        System.out.println("----------------------------");
        System.out.println("Innlesning av oppgave-figurer");
        System.out.println("----------------------------");
        System.out.println("Skriv inn paramterne som" +
         " gjelder for");
        System.out.println("de tre objektene i oppgaven." +
```

```
     " Når det");
    System.out.println("spørres etter en paramter, " +
     "skriv inn");
    System.out.println("verdien til paramteren til " +
     "hver av de");
    System.out.println("tre objektene (et objekt per " +
     "figur).");
    System.out.println("");

    System.out.print("Skriv inn x-koordinater på " +
     "objektene (3 heltall, " +
     "adskilt av mellomrom, avsluttet med enter): ");
    try {
        inline = in.readLine();
    } catch (IOException e) { }
    if (inline != null) {
        String[] splitted = inline.split(" ");
        try {
            for (int i = 0; i < splitted.length; i++) {
                x[i] = Integer.parseInt(splitted[i]);
            }
        } catch (NumberFormatException e) { }
    }

    System.out.print("Skriv inn y-koordinater på " +
     "objektene (3 heltall, " +
"adskilt av mellomrom, avsluttet med enter): ");
    try {
        inline = in.readLine();
    } catch (IOException e) { }
    if (inline != null) {
        String[] splitted = inline.split(" ");
        try {
            for (int i = 0; i < splitted.length; i++) {
                y[i] = Integer.parseInt(splitted[i]);
            }
        } catch (NumberFormatException e) { }
    }

    System.out.print("Skriv inn bredde på objektene " +
     "(3 heltall, " +
    "adskilt av mellomrom, avsluttet med enter): ");
    try {
```

```
        inline = in.readLine();
    } catch (IOException e) { }
    if (inline != null) {
        String[] splitted = inline.split(" ");
        try {
            for (int i = 0; i < splitted.length; i++) {
                width[i] =
                    Integer.parseInt(splitted[i]);
            }
        } catch (NumberFormatException e) { }
    }

    System.out.print("Skriv inn høyde på objektene " +
     "(3 heltall, " +
    "adskilt av mellomrom, avsluttet med enter): ");
    try {
        inline = in.readLine();
    } catch (IOException e) { }
    if (inline != null) {
        String[] splitted = inline.split(" ");
        try {
            for (int i = 0; i < splitted.length; i++) {
                height[i] =
                    Integer.parseInt(splitted[i]);
            }
        } catch (NumberFormatException e) { }
    }

    System.out.print("Skriv inn farge på objektene " +
     "(3 heltall, " +
    "adskilt av mellomrom, avsluttet med enter 0 = " +
    "sort, 1 = hvit): ");
    try {
        inline = in.readLine();
    } catch (IOException e) { }
    if (inline != null) {
        String[] splitted = inline.split(" ");
        try {
            for (int i = 0; i < splitted.length; i++) {
                color[i] = Integer.parseInt(splitted[i]);
            }
        } catch (NumberFormatException e) { }
    }
```

```
System.out.print("Skriv inn objektrotasjon på " +
 "objektene (3 heltall, " +
"adskilt av mellomrom, avsluttet med enter): ");
try {
    inline = in.readLine();
} catch (IOException e) { }
if (inline != null) {
    String[] splitted = inline.split(" ");
    try {
        for (int i = 0; i < splitted.length; i++) {
            objrotation[i] =
                Integer.parseInt(splitted[i]);
        }
    } catch (NumberFormatException e) { }
}

//---------------------------------------------

System.out.println("");
System.out.println("----------------------" +
 "--------");
System.out.println("Innlesning av " +
 "svaralternativer");
System.out.println("----------------------" +
 "--------");
System.out.println("Skriv inn paramterne som " +
 "gjelder for");
System.out.println("de fire objektene i " +
 "svaralternativene.");
System.out.println("Når det spørres etter en " +
 "paramter,");
System.out.println("skriv inn verdien til " +
 "paramteren til hver");
System.out.println("av de fire objektene (et " +
 "objekt per");
System.out.println("løsningalternativ).");
System.out.println("");

System.out.print("Skriv inn x-koordinater på " +
 "objektene (4 heltall, " +
"adskilt av mellomrom, avsluttet med enter): ");
try {
```

```
        inline = in.readLine();
    } catch (IOException e) { }
    if (inline != null) {
        String[] splitted = inline.split(" ");
        try {
            for (int i = 0; i < splitted.length; i++) {
               losx[i] = Integer.parseInt(splitted[i]);
            }
        } catch (NumberFormatException e) { }
    }

    System.out.print("Skriv inn y-koordinater på " +
     "objektene (4 heltall, " +
    "adskilt av mellomrom, avsluttet med enter): ");
    try {
        inline = in.readLine();
    } catch (IOException e) { }
    if (inline != null) {
        String[] splitted = inline.split(" ");
        try {
            for (int i = 0; i < splitted.length; i++) {
                losy[i] = Integer.parseInt(splitted[i]);
            }
        } catch (NumberFormatException e) { }
    }

    System.out.print("Skriv inn bredde på objektene " +
     "(4 heltall, " +
    "adskilt av mellomrom, avsluttet med enter): ");
    try {
        inline = in.readLine();
    } catch (IOException e) { }
    if (inline != null) {
        String[] splitted = inline.split(" ");
        try {
            for (int i = 0; i < splitted.length; i++) {
                loswidth[i] =
                    Integer.parseInt(splitted[i]);
            }
        } catch (NumberFormatException e) { }
    }

    System.out.print("Skriv inn høyde på objektene " +
```

```
 "(4 heltall, " +
"adskilt av mellomrom, avsluttet med enter): ");
try {
    inline = in.readLine();
} catch (IOException e) { }
if (inline != null) {
    String[] splitted = inline.split(" ");
    try {
        for (int i = 0; i < splitted.length; i++) {
            losheight[i] =
                Integer.parseInt(splitted[i]);
        }
    } catch (NumberFormatException e) { }
}

System.out.print("Skriv inn farge på objektene " +
 "(4 heltall, " +
"adskilt av mellomrom, avsluttet med enter 0 = " +
"sort, 1 = hvit): ");
try {
    inline = in.readLine();
} catch (IOException e) { }
if (inline != null) {
    String[] splitted = inline.split(" ");
    try {
        for (int i = 0; i < splitted.length; i++) {
            loscolor[i] =
                Integer.parseInt(splitted[i]);
        }
    } catch (NumberFormatException e) { }
}

System.out.print("Skriv inn objektrotasjon på " +
 "objektene (4 heltall, " +
"adskilt av mellomrom, avsluttet med enter): ");
try {
    inline = in.readLine();
} catch (IOException e) { }
if (inline != null) {
    String[] splitted = inline.split(" ");
    try {
        for (int i = 0; i < splitted.length; i++) {
            losobjrotation[i] =
```

```
                        Integer.parseInt(splitted[i]);
                }
            } catch (NumberFormatException e) { }
        }

        //Regner ut figurrotasjon og midtavstand
        for (int i = 0; i < 3; i++) {
            middledist[i] =
                (int)(Math.sqrt(x[i]*x[i]+y[i]*y[i])+0.5);
            figrotation[i] =
                (int)(Math.toDegrees(Math.atan2(
                        (double)x[i], (double)y[i])));

            if (figrotation[i] < 0) {
                figrotation[i] += 360;
            }

            if (objrotation[i] < 0) {
                objrotation[i] += 360;
            }
        }

        //Justerer figurrotasjon og objektrotasjon
        for (int i = 0; i < 4; i++) {
            losmiddledist[i] =
                (int)(Math.sqrt(losx[i]*losx[i]+
                        losy[i]*losy[i])+0.5);
            losfigrotation[i] =
                (int)(Math.toDegrees(Math.atan2(
                        (double)losx[i], (double)losy[i])));

            if (losfigrotation[i] < 0) {
                losfigrotation[i] += 360;
            }

            if (losobjrotation[i] < 0) {
                losobjrotation[i] += 360;
            }
        }

        justerFigurrotasjon();
        justerObjektrotasjon();
    }
```

```
/**
 * Klassen justerer figurrotasjon som beskrevet i
 * kapittel 6.3.2.1 i rapporten.
 */
public void justerFigurrotasjon() {
    if (figrotation[0] != figrotation[2]) {
        //den i midten er minst
        if (figrotation[1] < figrotation[0] &&
                figrotation[1] < figrotation[2]) {
            if (figrotation[0] > figrotation[2]) {
                figrotation[1]+= 360;
                figrotation[2]+= 360;
            } else {
                figrotation[0]+= 360;
                figrotation[1]+= 360;
            }
        }

        //den i midten er størst
        if (figrotation[1] > figrotation[0] &&
                figrotation[1] > figrotation[2]) {
            if (figrotation[0] > figrotation[2]) {
                figrotation[2]+= 360;
            } else {
                figrotation[0]+= 360;
            }
        }
    }
}

/**
 * Klassen justerer objektrotasjon som beskrevet i
 * kapittel 6.3.2.4 i rapporten.
 */
public void justerObjektrotasjon() {
    if (objrotation[0] != objrotation[2]) {
        //den i midten er minst
        if (objrotation[1] < objrotation[0] &&
                objrotation[1] < objrotation[2]) {
            if (objrotation[0] > objrotation[2]) {
                objrotation[1]+= 360;
                objrotation[2]+= 360;
```

```
                } else {
                    objrotation[0]+= 360;
                    objrotation[1]+= 360;
                }
            }

            //den i midten er størst
            if (objrotation[1] > objrotation[0] &&
                    objrotation[1] > objrotation[2]) {
                if (objrotation[0] > objrotation[2]) {
                    objrotation[2]+= 360;
                } else {
                    objrotation[0]+= 360;
                }
            }
        }
    }

    public void evaluate(EvolutionState state,
            Individual ind, int threadnum) {
        int hits = 0;
        int sum = 0;
        int expectedResult;
        int result;

        //------------------------------------------------- X
        if (step == 1 && x[0] == x[2]) {
            solx = x[1];
            xfound = true;
            step = 2;
            hits = 0;
            sum = 0;
        } else if (step == 1 && gen < genbase * 1) {
            gen++;
        currentX = x[0];
        expectedResult = x[1];

        ((GPIndividual)ind).trees[0].child.eval(
                state, threadnum, input, stack,
                ((GPIndividual)ind), this);
        result = Math.abs(expectedResult - input.x);
        if (result == 0) {
            hits++;
```

```
        }
        sum += result;

        //--------

        currentX = x[1];
        expectedResult = x[2];

        ((GPIndividual)ind).trees[0].child.eval(
                state, threadnum, input, stack,
                ((GPIndividual)ind), this);
        result = Math.abs(expectedResult - input.x);
        if (result == 0) {
            hits++;
        }
        sum += result;

        // the fitness better be KozaFitness!
        if (sum < 0) {
            sum = 10000;
        }
        KozaFitness f1 = ((KozaFitness)ind.fitness);
        f1.setFitness(state, (float)sum+1);
        f1.hits = hits;
        ind.evaluated = true;

        if (f1.hits == 2) {
            currentX = x[2];

            ((GPIndividual)ind).trees[0].child.eval(
                    state,threadnum,input,stack,
                    ((GPIndividual)ind),this);

            solx = input.x;
            xfound = true;

            step = 2;
            hits = 0;
            sum = 0;
        }
        } else if (step == 1) {
            step = 2;
        KozaFitness f1 = ((KozaFitness)ind.fitness);
```

```
f1.setFitness(state, (float)1000);
f1.hits = 0;
ind.evaluated = true;

xfound = false;;
}

//------------------------------------------------- Y
if (step == 2 && y[0] == y[2]) {
    soly = y[1];
    yfound = true;
    step = 3;
    hits = 0;
    sum = 0;
} else if (step == 2 && gen < genbase * 2) {
    gen++;
currentX = y[0];
expectedResult = y[1];

((GPIndividual)ind).trees[0].child.eval(
        state, threadnum, input, stack,
        ((GPIndividual)ind), this);
result = Math.abs(expectedResult - input.x);
if (result == 0) {
    hits++;
}
sum += result;

//--------

currentX = y[1];
expectedResult = y[2];

((GPIndividual)ind).trees[0].child.eval(
        state, threadnum, input, stack,
        ((GPIndividual)ind), this);
result = Math.abs(expectedResult - input.x);
if (result == 0) {
    hits++;
}
sum += result;

// the fitness better be KozaFitness!
```

```
if (sum < 0) {
    sum = 10000;
}
KozaFitness f2 = ((KozaFitness)ind.fitness);
f2.setFitness(state, (float)sum+1);
f2.hits = hits;
ind.evaluated = true;

if (f2.hits == 2) {
    currentX = y[2];

    ((GPIndividual)ind).trees[0].child.eval(
            state,threadnum,input,stack,
            ((GPIndividual)ind),this);
    soly = input.x;
    yfound = true;

    step = 3;
    hits = 0;
    sum = 0;
}
} else if (step == 2) {
    step = 3;
KozaFitness f2 = ((KozaFitness)ind.fitness);
f2.setFitness(state, (float)1000);
f2.hits = 0;
ind.evaluated = true;

yfound = false;
}

//------------------------------------------- Bredde
if (step == 3 && width[0] == width[2]) {
    solwidth = width[1];
    widthfound = true;
    step = 4;
    hits = 0;
    sum = 0;
} else if (step == 3 && gen < genbase * 3) {
    gen++;
currentX = width[0];
expectedResult = width[1];
```

```
((GPIndividual)ind).trees[0].child.eval(
        state, threadnum, input, stack,
        ((GPIndividual)ind), this);
result = Math.abs(expectedResult - input.x);
if (result == 0) {
    hits++;
}
sum += result;

//--------

currentX = width[1];
expectedResult = width[2];

((GPIndividual)ind).trees[0].child.eval(
        state, threadnum, input, stack,
        ((GPIndividual)ind), this);
result = Math.abs(expectedResult - input.x);
if (result == 0) {
    hits++;
}
sum += result;

// the fitness better be KozaFitness!
if (sum < 0) {
    sum = 10000;
}
KozaFitness f3 = ((KozaFitness)ind.fitness);
f3.setFitness(state, (float)sum+1);
f3.hits = hits;
ind.evaluated = true;

if (f3.hits == 2) {
    currentX = width[2];

    ((GPIndividual)ind).trees[0].child.eval(
            state,threadnum,input,stack,
            ((GPIndividual)ind),this);
    solwidth = input.x;
    widthfound = true;

    step = 4;
    hits = 0;
```

```
    sum = 0;
}
} else if (step == 3) {
    step = 4;
KozaFitness f4 = ((KozaFitness)ind.fitness);
f4.setFitness(state, (float)1000);
f4.hits = 0;
ind.evaluated = true;

widthfound = false;;
}

//----------------------------------------------- Høyde
if (step == 4 && height[0] == height[2]) {
    solheight = height[1];
    heightfound = true;
    step = 5;
    hits = 0;
    sum = 0;
} else if (step == 4 && gen < genbase * 4) {
    gen++;
currentX = height[0];
expectedResult = height[1];

((GPIndividual)ind).trees[0].child.eval(
        state, threadnum, input, stack,
        ((GPIndividual)ind), this);
result = Math.abs(expectedResult - input.x);
if (result == 0) {
    hits++;
}
sum += result;

//--------

currentX = height[1];
expectedResult = height[2];

((GPIndividual)ind).trees[0].child.eval(
        state, threadnum, input, stack,
        ((GPIndividual)ind), this);
result = Math.abs(expectedResult - input.x);
if (result == 0) {
```

```
        hits++;
    }
    sum += result;

    // the fitness better be KozaFitness!
    if (sum < 0) {
        sum = 10000;
    }
    KozaFitness f4 = ((KozaFitness)ind.fitness);
    f4.setFitness(state, (float)sum+1);
    f4.hits = hits;
    ind.evaluated = true;

    if (f4.hits == 2) {
        currentX = height[2];

        ((GPIndividual)ind).trees[0].child.eval(
                state,threadnum,input,stack,
                ((GPIndividual)ind),this);
        solheight = input.x;
        heightfound = true;

        step = 5;
        hits = 0;
        sum = 0;
    }
    } else if (step == 4) {
        step = 5;
    KozaFitness f5 = ((KozaFitness)ind.fitness);
    f5.setFitness(state, (float)1000);
    f5.hits = 0;
    ind.evaluated = true;

    heightfound = false;;
    }

    //------------------------------------- Figurrotasjon
    if (step == 5 &&
            figrotation[0] == figrotation[2]) {
        solfigrotation = figrotation[1];
        figrotationfound = true;
        step = 6;
        hits = 0;
```

```
    sum = 0;
} else if (step == 5 && gen < genbase * 5) {
    gen++;
    currentX = figrotation[0];
    expectedResult = figrotation[1];

((GPIndividual)ind).trees[0].child.eval(
        state, threadnum, input, stack,
        ((GPIndividual)ind), this);
result = Math.abs(expectedResult - input.x);
if (result == 0) {
    hits++;
}
sum += result;

//--------

currentX = figrotation[1];
expectedResult = figrotation[2];

((GPIndividual)ind).trees[0].child.eval(
        state, threadnum, input, stack,
        ((GPIndividual)ind), this);
result = Math.abs(expectedResult - input.x);
if (result == 0) {
    hits++;
}
sum += result;

// the fitness better be KozaFitness!
if (sum < 0) {
    sum = 10000;
}
KozaFitness f5 = ((KozaFitness)ind.fitness);
f5.setFitness(state, (float)sum+1);
f5.hits = hits;
ind.evaluated = true;

if (f5.hits == 2) {
    currentX = figrotation[2];

    ((GPIndividual)ind).trees[0].child.eval(
            state,threadnum,input,stack,
```

```
                ((GPIndividual)ind),this);
        solfigrotation = input.x;
        figrotationfound = true;

        step = 6;
        hits = 0;
        sum = 0;
    }
    } else if (step == 5) {
        step = 6;
    KozaFitness f6 = ((KozaFitness)ind.fitness);
    f6.setFitness(state, (float)1000);
    f6.hits = 0;
    ind.evaluated = true;

    figrotationfound = false;;
    }

    //--------------------------------------- Midtavstand
    if (step == 6 && middledist[0] == middledist[2]) {
        solmiddeldist = middledist[1];
        middeldistfound = true;

        step = 7;
        hits = 0;
        sum = 0;
    } else if (step == 6 && gen < genbase * 6) {
        gen++;
    currentX = middledist[0];
    expectedResult = middledist[1];

    ((GPIndividual)ind).trees[0].child.eval(
            state, threadnum, input, stack,
            ((GPIndividual)ind), this);
    result = Math.abs(expectedResult - input.x);
    if (result == 0) {
        hits++;
    }
    sum += result;

    //--------

    currentX = middledist[1];
```

```
        expectedResult = middledist[2];

        ((GPIndividual)ind).trees[0].child.eval(
                state, threadnum, input, stack,
                ((GPIndividual)ind), this);
        result = Math.abs(expectedResult - input.x);
        if (result == 0) {
            hits++;
        }
        sum += result;

        // the fitness better be KozaFitness!
        if (sum < 0) {
            sum = 10000;
        }
        KozaFitness f6 = ((KozaFitness)ind.fitness);
        f6.setFitness(state, (float)sum+1);
        f6.hits = hits;
        ind.evaluated = true;

        if (f6.hits == 2) {
            currentX = middledist[2];

            ((GPIndividual)ind).trees[0].child.eval(
                    state,threadnum,input,stack,
                    ((GPIndividual)ind),this);
            solmiddeldist = input.x;
            middeldistfound = true;

            step = 7;
            hits = 0;
            sum = 0;
        }
        } else if (step == 6) {
//              System.out.println("X ikke funnet!");
            step = 7;
        KozaFitness f7 = ((KozaFitness)ind.fitness);
        f7.setFitness(state, (float)1000);
        f7.hits = 0;
        ind.evaluated = true;

        middeldistfound = false;;
        }
```

```
//----------------------------------------------- Farge
if (step == 7) {
    if (color[0] == color[2]) {
        solcolor = color[1];
        colorfound = true;
    } else {
        solcolor = 1-color[1];
        colorfound = true;
    }
    step = 8;
    hits = 0;
    sum = 0;
}

//------------------------------------- Objektrotasjon
if (step == 8 &&
        objrotation[0] == objrotation[2]) {
    solobjrotation = objrotation[1];
    objrotationfound = true;
    step = 9;
    hits = 0;
    sum = 0;
} else if (step == 8 && gen < genbase * 7) {
currentX = objrotation[0];
expectedResult = objrotation[1];

((GPIndividual)ind).trees[0].child.eval(
        state, threadnum, input, stack,
        ((GPIndividual)ind), this);
result = Math.abs(expectedResult - input.x);
if (result == 0) {
    hits++;
}
sum += result;

//--------

currentX = objrotation[1];
expectedResult = objrotation[2];

((GPIndividual)ind).trees[0].child.eval(
        state, threadnum, input, stack,
```

```
        ((GPIndividual)ind), this);
result = Math.abs(expectedResult - input.x);
if (result == 0) {
    hits++;
}
sum += result;

// the fitness better be KozaFitness!
if (sum < 0) {
    sum = 10000;
}
KozaFitness f8 = ((KozaFitness)ind.fitness);
f8.setFitness(state, (float)sum+1);
f8.hits = hits;
ind.evaluated = true;

if (f8.hits == 2) {
    currentX = objrotation[2];

    ((GPIndividual)ind).trees[0].child.eval(
            state,threadnum,input,stack,
            ((GPIndividual)ind),this);
        solobjrotation = input.x;
        objrotationfound = true;

    step = 9;
    hits = 0;
    sum = 0;
}
} else if (step == 8) {
    step = 9;
KozaFitness f8 = ((KozaFitness)ind.fitness);
f8.setFitness(state, (float)1000);
f8.hits = 0;
ind.evaluated = true;

objrotationfound = false;;
}

//Alle stegene er ferdige. Skriv ut resultater og
//avslutt.
if (step == 9) {
    while (solfigrotation >= 360) {
```

```
            solfigrotation -= 360;
        }

        while (solfigrotation < 0) {
            solfigrotation += 360;
        }

        while (solobjrotation >= 360) {
            solobjrotation -= 360;
        }

        while (solobjrotation < 0) {
            solobjrotation += 360;
        }

        System.out.println("");
        System.out.println("------- LØSNING -------");
        System.out.println("Beregninger:");

        if (!(xfound && yfound) &&
                (figrotationfound &&
                        middeldistfound)) {
            System.out.println("Figurrotasjon: " +
                    solfigrotation);
            System.out.println("Midtavstand: " +
                    solmiddeldist);

            if (solfigrotation ==
                losfigrotation[0]) {
                expectedAltA++;
            }

            if (solfigrotation ==
                losfigrotation[1]) {
                expectedAltB++;
            }

            if (solfigrotation ==
                losfigrotation[2]) {
                expectedAltC++;
            }

            if (solfigrotation ==
```

```
                losfigrotation[3]) {
                expectedAltD++;
            }

            if (solmiddeldist == losmiddledist[0]) {
                expectedAltA++;
            }

            if (solmiddeldist == losmiddledist[1]) {
                expectedAltB++;
            }

            if (solmiddeldist == losmiddledist[2]) {
                expectedAltC++;
            }

            if (solmiddeldist == losmiddledist[3]) {
                expectedAltD++;
            }
        } else if (!(figrotationfound &&
                middeldistfound) &&
                (xfound && yfound)) {
            int f = (int)(Math.toDegrees(Math.atan2(
                    (double)solx, (double)soly)));
            int m = (int)(Math.sqrt(solx*solx+soly*soly)+
                    0.5);

            System.out.println("Figurrotasjon: " + f);
            System.out.println("Midtavstand: " + m);

            if (f == losfigrotation[0]) {
                expectedAltA++;
            }

            if (f == losfigrotation[1]) {
                expectedAltB++;
            }

            if (f == losfigrotation[2]) {
                expectedAltC++;
            }

            if (f == losfigrotation[3]) {
```

```
                expectedAltD++;
            }

            if (m == losmiddledist[0]) {
                expectedAltA++;
            }

            if (m == losmiddledist[1]) {
                expectedAltB++;
            }

            if (m == losmiddledist[2]) {
                expectedAltC++;
            }

            if (m == losmiddledist[3]) {
                expectedAltD++;
            }
        } else if (!(xfound && yfound) &&
                !(figrotationfound &&
                        middeldistfound)) {
            System.out.println("Klarte ikke å plassere" +
             " objekt!");
        } else {
            int solmidtxy = (int)(Math.sqrt(solx*solx+
                    soly*soly)+0.5);
            int solgradxy = (int)(Math.toDegrees(
                    Math.atan2((double)solx,
                            (double)soly)));

            if (solgradxy < 0) {
                solgradxy += 360;
            }

            int leastalt = 0;
            int leastdist = 100000;

            if (Math.abs(solmidtxy - losmiddledist[0]) +
                    Math.abs(solgradxy -
                            losfigrotation[0]) <
                            leastdist) {
                leastdist = Math.abs(solmidtxy -
                        losmiddledist[0]) +
```

```
            Math.abs(solgradxy -
                    losfigrotation[0]);
    leastalt = 1;
}

if (Math.abs(solfigrotation -
        losmiddledist[0]) +
        Math.abs(solmiddeldist -
                losfigrotation[0]) <
                leastdist) {
    leastdist = Math.abs(solfigrotation
            - losmiddledist[0]) + Math.abs(
                    solmiddeldist -
                    losfigrotation[0]);
    leastalt = 2;
}

if (Math.abs(solmidtxy - losmiddledist[1]) +
        Math.abs(solgradxy -
                losfigrotation[1]) <
                leastdist) {
    leastdist = Math.abs(solmidtxy -
            losmiddledist[1]) + Math.abs(
                    solgradxy -
                    losfigrotation[1]);
    leastalt = 3;
}

if (Math.abs(solfigrotation -
        losmiddledist[1]) + Math.abs(
                solmiddeldist -
                losfigrotation[1]) <
                leastdist) {
    leastdist = Math.abs(solfigrotation
            - losmiddledist[1]) + Math.abs(
                    solmiddeldist -
                    losfigrotation[1]);
    leastalt = 4;
}

if (Math.abs(solmidtxy - losmiddledist[2]) +
        Math.abs(solgradxy -
                losfigrotation[2]) <
```

```
                leastdist) {
        leastdist = Math.abs(solmidtxy -
                losmiddledist[2]) + Math.abs(
                        solgradxy -
                        losfigrotation[2]);
        leastalt = 5;
    }

    if (Math.abs(solfigrotation -
            losmiddledist[2]) + Math.abs(
                solmiddeldist -
                losfigrotation[2]) <
                leastdist) {
        leastdist = Math.abs(solfigrotation
                - losmiddledist[2]) + Math.abs(
                        solmiddeldist -
                        losfigrotation[2]);
        leastalt = 6;
    }

    if (Math.abs(solmidtxy - losmiddledist[3]) +
            Math.abs(solgradxy -
                losfigrotation[3]) <
                leastdist) {
        leastdist = Math.abs(solmidtxy -
                losmiddledist[3]) + Math.abs(
                        solgradxy -
                        losfigrotation[3]);
        leastalt = 7;
    }

    if (Math.abs(solfigrotation -
            losmiddledist[3]) + Math.abs(
                solmiddeldist -
                losfigrotation[3]) <
                leastdist) {
        leastdist = Math.abs(solfigrotation
                - losmiddledist[3]) + Math.abs(
                        solmiddeldist -
                        losfigrotation[3]);
        leastalt = 8;
    }
```

```
if (leastalt % 2 == 0) {
    System.out.println("Figurrotasjon: " +
            solfigrotation);
    System.out.println("Midtavstand: " +
            solmiddeldist);
    if (solfigrotation ==
        losfigrotation[0]) {
        expectedAltA++;
    }

    if (solfigrotation ==
        losfigrotation[1]) {
        expectedAltB++;
    }

    if (solfigrotation ==
        losfigrotation[2]) {
        expectedAltC++;
    }

    if (solfigrotation ==
        losfigrotation[3]) {
        expectedAltD++;
    }

    if (solmiddeldist ==
        losmiddledist[0]) {
        expectedAltA++;
    }

    if (solmiddeldist ==
        losmiddledist[1]) {
        expectedAltB++;
    }

    if (solmiddeldist ==
        losmiddledist[2]) {
        expectedAltC++;
    }

    if (solmiddeldist ==
        losmiddledist[3]) {
        expectedAltD++;
```

```
            }
        } else {
            System.out.println("Figurrotasjon: " +
                    solgradxy);
            System.out.println("Midtavstand: " +
                    solmidtxy);

            if (solgradxy == losfigrotation[0]) {
                expectedAltA++;
            }

            if (solgradxy == losfigrotation[1]) {
                expectedAltB++;
            }

            if (solgradxy == losfigrotation[2]) {
                expectedAltC++;
            }

            if (solgradxy == losfigrotation[3]) {
                expectedAltD++;
            }

            if (solmidtxy == losmiddledist[0]) {
                expectedAltA++;
            }

            if (solmidtxy == losmiddledist[1]) {
                expectedAltB++;
            }

            if (solmidtxy == losmiddledist[2]) {
                expectedAltC++;
            }

            if (solmidtxy == losmiddledist[3]) {
                expectedAltD++;
            }
        }
    }

    if (widthfound) {
        System.out.println("Bredde: " + solwidth);
```

```
        if (solwidth == loswidth[0]) {
            expectedAltA++;
        }

        if (solwidth == loswidth[1]) {
            expectedAltB++;
        }

        if (solwidth == loswidth[2]) {
            expectedAltC++;
        }

        if (solwidth == loswidth[3]) {
            expectedAltD++;
        }
    } else {
        System.out.println("Bredde ikke funnet!");
        solwidth = 0;
    }

    if (heightfound) {
        System.out.println("Høyde: " + solheight);
        if (solheight == losheight[0]) {
            expectedAltA++;
        }

        if (solheight == losheight[1]) {
            expectedAltB++;
        }

        if (solheight == losheight[2]) {
            expectedAltC++;
        }

        if (solheight == losheight[3]) {
            expectedAltD++;
        }
    } else {
        System.out.println("Høyde ikke funnet!");
        solheight = 0;
    }

    if (colorfound) {
```

```
        System.out.println("Farge: " + solcolor);
        if (solcolor == loscolor[0]) {
            expectedAltA++;
        }

        if (solcolor == loscolor[1]) {
            expectedAltB++;
        }

        if (solcolor == loscolor[2]) {
            expectedAltC++;
        }

        if (solcolor == loscolor[3]) {
            expectedAltD++;
        }
    } else {
        System.out.println("Farge ikke funnet!");
        solcolor = 0;
    }

    if (objrotationfound) {
        System.out.println("Objektrotasjon: " +
                solobjrotation);
        if (solobjrotation ==
            losobjrotation[0]) {
            expectedAltA++;
        }

        if (solobjrotation ==
            losobjrotation[1]) {
            expectedAltB++;
        }

        if (solobjrotation ==
            losobjrotation[2]) {
            expectedAltC++;
        }

        if (solobjrotation ==
            losobjrotation[3]) {
            expectedAltD++;
        }
```

```
} else {
    System.out.println("Objektrotasjon ikke " +
     "funnet!");
    solobjrotation = 0;
}

System.out.println("");
System.out.println("Svaralternativ A matcher" +
 " på " + expectedAltA + " av 6 målte" +
 " elementer.");
System.out.println("Svaralternativ B matcher" +
 " på " + expectedAltB + " av 6 målte" +
 " elementer.");
System.out.println("Svaralternativ C matcher" +
 " på " + expectedAltC + " av 6 målte" +
 " elementer.");
System.out.println("Svaralternativ D matcher" +
 " på " + expectedAltD + " av 6 målte" +
 " elementer.");
System.out.println("");

if (expectedAltA >= expectedAltB &&
        expectedAltA >= expectedAltC &&
        expectedAltA >= expectedAltD) {
    System.out.println("Vi velger derfor" +
     " svaralternativ A.");
} else if (expectedAltB >= expectedAltA &&
        expectedAltB >= expectedAltC &&
        expectedAltB >= expectedAltD) {
    System.out.println("Vi velger derfor" +
     " svaralternativ B.");
} else if (expectedAltC >= expectedAltA &&
        expectedAltC >= expectedAltB &&
        expectedAltC >= expectedAltD) {
    System.out.println("Vi velger derfor" +
     " svaralternativ C.");
} else if (expectedAltD >= expectedAltA &&
        expectedAltD >= expectedAltB &&
        expectedAltD >= expectedAltC) {
    System.out.println("Vi velger derfor" +
     " svaralternativ D.");
}
```

```
            System.exit(0);
        }
    }

    public void describe(Individual ind,
            EvolutionState state,
            int threadnum,
            int log,
            int verbosity) {
        //Brukes ikke
    }
}
```

D.4 Fourtyfive.java

```
package ec.app.figure;

import ec.EvolutionState;
import ec.Problem;
import ec.gp.ADFStack;
import ec.gp.GPData;
import ec.gp.GPIndividual;
import ec.gp.GPNode;
import ec.util.Parameter;

/**
 * Klassen er en konstant (node) i GP-treet.
 *
 * @author Anders Lund Fredriksen <anderf@stud.ntnu.no>
 */
public class Fourtyfive extends GPNode {
    public String toString() {
        return "ONE";
    }

    public void checkConstraints(final EvolutionState state,
            final int tree,
            final GPIndividual typicalIndividual,
            final Parameter individualBase) {
        super.checkConstraints(state, tree, typicalIndividual,
```

```
                individualBase);
        if (children.length != 0)
            state.output.error(
                    "Incorrect number of children for node " +
                    toStringForError() + " at " +
                    individualBase);
        }

    public void eval(final EvolutionState state,
                     final int thread,
                     final GPData input,
                     final ADFStack stack,
                     final GPIndividual individual,
                     final Problem problem) {
        FigureData rd = ((FigureData)(input));
        rd.x = 45;
    }
}
```

D.5 Mul.java

```
package ec.app.figure;

import ec.EvolutionState;
import ec.Problem;
import ec.gp.ADFStack;
import ec.gp.GPData;
import ec.gp.GPIndividual;
import ec.gp.GPNode;
import ec.util.Parameter;

/**
 * Klassen er en funksjon (node) i GP-treet. Funksjonen
 * multipliserer to tall.
 *
 * @author Anders Lund Fredriksen <anderf@stud.ntnu.no>
 */
public class Mul extends GPNode {
    public String toString() {
        return "*";
    }
```

```
    public void checkConstraints(final EvolutionState state,
            final int tree,
            final GPIndividual typicalIndividual,
            final Parameter individualBase) {
        super.checkConstraints(state, tree, typicalIndividual,
                individualBase);
        if (children.length != 2)
            state.output.error(
                    "Incorrect number of children for node " +
                    toStringForError() + " at " +
                    individualBase);
    }

    public void eval(final EvolutionState state,
                     final int thread,
                     final GPData input,
                     final ADFStack stack,
                     final GPIndividual individual,
                     final Problem problem) {
        int result;
        FigureData rd = ((FigureData)(input));

        children[0].eval(state, thread, input, stack,
                individual, problem);
        result = rd.x;

        children[1].eval(state, thread, input, stack,
                individual, problem);
        int t = rd.x;
        rd.x = result * t;
    }
}
```

D.6 One.java

```
package ec.app.figure;

import ec.EvolutionState;
import ec.Problem;
import ec.gp.ADFStack;
```

```
import ec.gp.GPData;
import ec.gp.GPIndividual;
import ec.gp.GPNode;
import ec.util.Parameter;

/**
 * Klassen er en konstant (node) i GP-treet.
 *
 * @author Anders Lund Fredriksen <anderf@stud.ntnu.no>
 */
public class One extends GPNode {
    public String toString() {
        return "ONE";
    }

    public void checkConstraints(final EvolutionState state,
            final int tree,
            final GPIndividual typicalIndividual,
            final Parameter individualBase) {
        super.checkConstraints(state, tree, typicalIndividual,
                individualBase);
        if (children.length != 0)
            state.output.error(
                    "Incorrect number of children for node " +
                    toStringForError() + " at " +
                    individualBase);
        }

    public void eval(final EvolutionState state,
                     final int thread,
                     final GPData input,
                     final ADFStack stack,
                     final GPIndividual individual,
                     final Problem problem) {
        FigureData rd = ((FigureData)(input));
        rd.x = 1;
    }
}
```

D.7 Sub.java

```
package ec.app.figure;

import ec.EvolutionState;
import ec.Problem;
import ec.gp.ADFStack;
import ec.gp.GPData;
import ec.gp.GPIndividual;
import ec.gp.GPNode;
import ec.util.Parameter;

/**
 * Klassen er en funksjon (node) i GP-treet. Funksjonen
 * subtraherer et tall fra et annet.
 *
 * @author Anders Lund Fredriksen <anderf@stud.ntnu.no>
 */
public class Sub extends GPNode {
    public String toString() {
        return "-";
    }

    public void checkConstraints(final EvolutionState state,
            final int tree,
            final GPIndividual typicalIndividual,
            final Parameter individualBase) {
        super.checkConstraints(state, tree, typicalIndividual,
                individualBase);
        if (children.length != 2)
            state.output.error(
                    "Incorrect number of children for node " +
                    toStringForError() + " at " +
                    individualBase);
}

    public void eval(final EvolutionState state,
                     final int thread,
                     final GPData input,
                     final ADFStack stack,
                     final GPIndividual individual,
                     final Problem problem) {
```

```
        int result;
        FigureData rd = ((FigureData)(input));

        children[0].eval(state, thread, input, stack,
                individual, problem);
        result = rd.x;

        children[1].eval(state, thread, input, stack,
                individual, problem);
        int t = rd.x;
        rd.x = result - t;

    }
}
```

D.8 Ten.java

```
package ec.app.figure;

import ec.EvolutionState;
import ec.Problem;
import ec.gp.ADFStack;
import ec.gp.GPData;
import ec.gp.GPIndividual;
import ec.gp.GPNode;
import ec.util.Parameter;

/**
 * Klassen er en konstant (node) i GP-treet.
 *
 * @author Anders Lund Fredriksen <anderf@stud.ntnu.no>
 */
public class Ten extends GPNode {
    public String toString() {
        return "TEN";
    }

    public void checkConstraints(final EvolutionState state,
            final int tree,
            final GPIndividual typicalIndividual,
            final Parameter individualBase) {
```

```
        super.checkConstraints(state, tree, typicalIndividual,
                individualBase);
        if (children.length != 0)
            state.output.error(
                    "Incorrect number of children for node " +
                    toStringForError() + " at " +
                    individualBase);
        }

    public void eval(final EvolutionState state,
                     final int thread,
                     final GPData input,
                     final ADFStack stack,
                     final GPIndividual individual,
                     final Problem problem) {
        FigureData rd = ((FigureData)(input));
        rd.x = 10;
    }
}
```

D.9 X.java

```
package ec.app.figure;

import ec.EvolutionState;
import ec.Problem;
import ec.gp.ADFStack;
import ec.gp.GPData;
import ec.gp.GPIndividual;
import ec.gp.GPNode;
import ec.util.Parameter;

/**
 * Klassen er en variabel (node) i GP-treet.
 *
 * @author Anders Lund Fredriksen <anderf@stud.ntnu.no>
 */
public class X extends GPNode {
    private static String name = "X";

    public String toString() {
```

```
        return name;
    }

    public void checkConstraints(final EvolutionState state,
            final int tree,
            final GPIndividual typicalIndividual,
            final Parameter individualBase) {
        super.checkConstraints(state, tree, typicalIndividual,
                individualBase);
        if (children.length != 0)
            state.output.error(
                    "Incorrect number of children for node " +
                    toStringForError() + " at " +
                    individualBase);
    }

    public void eval(final EvolutionState state,
                     final int thread,
                     final GPData input,
                     final ADFStack stack,
                     final GPIndividual individual,
                     final Problem problem) {
        FigureData fd = ((FigureData)(input));
        fd.x = ((Figures)problem).currentX;
    }
}
```

D.10 figures.params

```
parent.0 = ../../gp/koza/koza.params

# Vi har ett funksjonssett av klasse GPFunctionSet.
gp.fs.size = 1
gp.fs.0 = ec.gp.GPFunctionSet

# Vi kaller funksjonssettet "f0".
# Det bruker standard GPFuncInfo-klasse.
gp.fs.0.name = f0
gp.fs.0.info = ec.gp.GPFuncInfo

# Vi har seks funksjoner i funksjonssettet. Disse er:
```

```
gp.fs.0.size = 7
gp.fs.0.func.0 = ec.app.figure.One
gp.fs.0.func.0.nc = nc0
gp.fs.0.func.1 = ec.app.figure.Add
gp.fs.0.func.1.nc = nc2
gp.fs.0.func.2 = ec.app.figure.Sub
gp.fs.0.func.2.nc = nc2
gp.fs.0.func.3 = ec.app.figure.Mul
gp.fs.0.func.3.nc = nc2
gp.fs.0.func.4 = ec.app.figure.X
gp.fs.0.func.4.nc = nc0
gp.fs.0.func.5 = ec.app.figure.Fourtyfive
gp.fs.0.func.5.nc = nc0
gp.fs.0.func.6 = ec.app.figure.Ten
gp.fs.0.func.6.nc = nc0

eval.problem = ec.app.figure.Figures
eval.problem.data = ec.app.figure.FigureData

# Følgende skal nesten alltid være samme som
# eval.problem.data
# Det definerer dataobjektet brukt internt i
# ADF-stakk-konteksten
eval.problem.stack.context.data = ec.app.figure.FigureData

breed.elite.0 = 3
generations = 1000
seed.0=time
```

Tillegg E

Kjøreeksempel, figurrekker

Under følger et kjøreeksempel der en av oppgavene fra Test nasjonen 2003 [NRK03, oppgave 27] er brukt som eksempel. Se readme.txt i rotkatalogen på den vedlagte CD-en for beskrivelse av hvordan man kjører programmet.

```
| ECJ
| An evolutionary computation system (version 12)
| Copyright 2004 by Sean Luke
| Contributors: L. Panait, G. Balan, Z. Skolicki, J. Bassett,
|               R. Hubley, and A. Chircop
| URL: http://cs.gmu.edu/~eclab/projects/ecj/
| Mail: sean@cs.gmu.edu
| Date: September 21, 2004
| Current Java: 1.5.0_02 / Java HotSpot(TM)
    Client VM-1.5.0_02-b09
| Required Minimum Java: 1.2


Threads:  breed/1 eval/1
Seed: 113993772
Setting up
Processing GP Types
Processing GP Node Constraints
Processing GP Function Sets
Processing GP Tree Constraints

-----------------------------
Innlesning av oppgave-figurer
-----------------------------
```

```
Skriv inn paramterne som gjelder for
de tre objektene i oppgaven. Når det
spørres etter en paramter, skriv inn
verdien til paramteren til hver av de
tre objektene (et objekt per figur).

Skriv inn x-koordinater på objektene (3 heltall, adskilt
av mellomrom, avsluttet med enter): -40 -40 0
Skriv inn y-koordinater på objektene (3 heltall, adskilt
av mellomrom, avsluttet med enter): 0 40 40
Skriv inn bredde på objektene (3 heltall, adskilt av
mellomrom, avsluttet med enter): 20 20 20
Skriv inn høyde på objektene (3 heltall, adskilt av
mellomrom, avsluttet med enter): 20 20 20
Skriv inn farge på objektene (3 heltall, adskilt av
mellomrom, avsluttet med enter 0 = sort, 1 = hvit): 1 0 1
Skriv inn objektrotasjon på objektene (3 heltall, adskilt
av mellomrom, avsluttet med enter): 0 0 0

------------------------------
Innlesning av svaralternativer
------------------------------
Skriv inn paramterne som gjelder for
de fire objektene i svaralternativene.
Når det spørres etter en paramter,
skriv inn verdien til paramteren til hver
av de fire objektene (et objekt per
svaralternativ).

Skriv inn x-koordinater på objektene (4 heltall, adskilt
av mellomrom, avsluttet med enter): 40 0 40 40
Skriv inn y-koordinater på objektene (4 heltall, adskilt
av mellomrom, avsluttet med enter): 40 40 0 40
Skriv inn bredde på objektene (4 heltall, adskilt av
mellomrom, avsluttet med enter): 20 20 20 20
Skriv inn høyde på objektene (4 heltall, adskilt av
mellomrom, avsluttet med enter): 20 20 20 20
Skriv inn farge på objektene (4 heltall, adskilt av
mellomrom, avsluttet med enter 0 = sort, 1 = hvit): 0 0 1 1
Skriv inn objektrotasjon på objektene (4 heltall, adskilt
av mellomrom, avsluttet med enter): 0 0 0 0
Initializing Generation 0
Generation 1
```

```
Generation 2
Generation 3
Generation 4
Generation 5
Generation 6
Generation 7
Generation 8
Generation 9

------- LØSNING -------
Beregninger:
Figurrotasjon: 45
Midtavstand: 57
Bredde: 20
Høyde: 20
Farge: 0
Objektrotasjon: 0

Svaralternativ A matcher på 6 av 6 målte elementer.
Svaralternativ B matcher på 4 av 6 målte elementer.
Svaralternativ C matcher på 3 av 6 målte elementer.
Svaralternativ D matcher på 5 av 6 målte elementer.

Vi velger derfor svaralternativ A.
Generation 10
```

Tillegg F

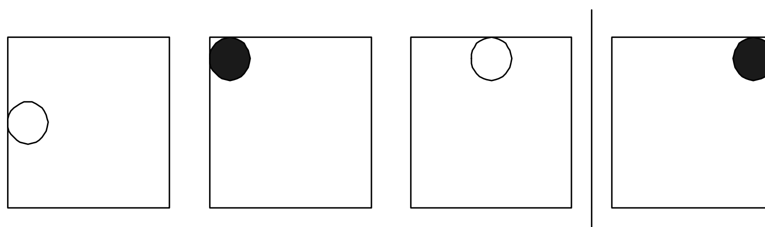
Resultater, figurrekker

Dette vedlegget inneholder resultater av kjøring av fem figuroppgaver. De fem oppgavene[1] finnes i figurene F.1, F.2, F.3, F.4 og F.5. Hver oppgave inneholder tre figurer som svarer til selve oppgaven (figurene er altså de tre første i en rekke, og oppgaven er å finne den fjerde), i tillegg til en figur som representerer fasiten.

Til hver oppgave er det også laget en oversikt over paramtre til å beskrive de ulike objektene i figurene. Som nevnt oversettes x- og y-koordinater til polare koordinater under kjøring. Inndataene er derfor basert på rektangulære koordinater, mens utdataene er basert på polare. Paramteroversiktene er gjengitt i tabellene F, F, F, F og F. Hver figur består av kun ett objekt.

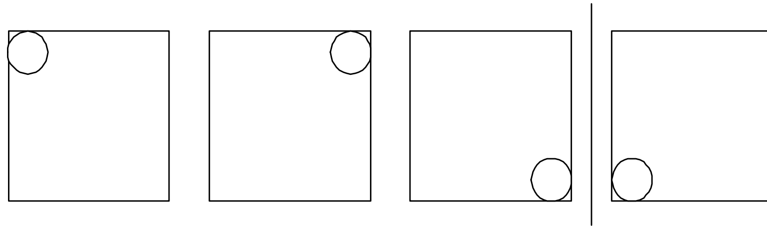
En oversikt over kjøreresultatene finnes i tabellene F, F, F, F og F. Hver oppgave er kjørt fem ganger uten svaralternativer, og med unntak av fjerde kjøring på femte oppgave (markert med **fet**) ble alle paramtrene evaluert riktig.

Videre diskusjon av resultatene finnes i kapittel 6.4.

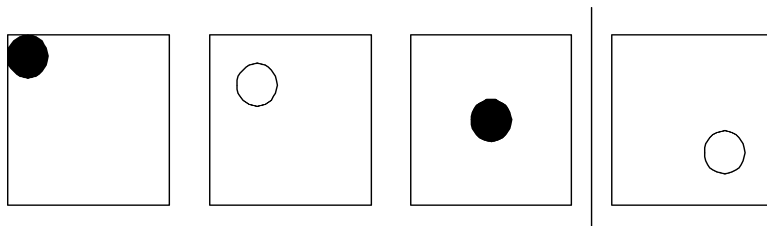


Figur F.1: Figurrekke brukt til testing, oppgave 1.

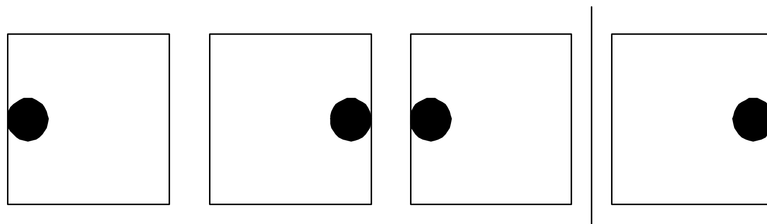
[1] Den første oppgaven er hentet fra Test nasjonen 2003 [NRK03, oppgave 27], mens resten av oppgavene er selvlagde.



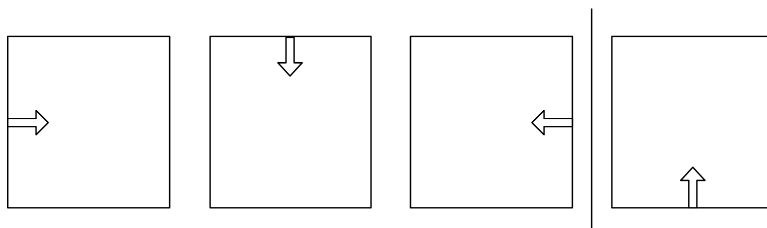
Figur F.2: Figurrekke brukt til testing, oppgave 2.



Figur F.3: Figurrekke brukt til testing, oppgave 3.



Figur F.4: Figurrekke brukt til testing, oppgave 4.



Figur F.5: Figurrekke brukt til testing, oppgave 5.

Tabell F.1: Variabler i figurrekke-oppgave 1.

Variabel	**Verdi 1**	**Verdi 2**	**Verdi 3**	**Ønsket utdata**
x	-40	-40	0	
y	0	40	40	
figurrotasjon				45
midtavstand				57
bredde	20	20	20	20
høyde	20	20	20	20
farge	1	0	1	0
objektrotasjon	0	0	0	0

Tabell F.2: Variabler i figurrekke-oppgave 2.

Variabel	**Verdi 1**	**Verdi 2**	**Verdi 3**	**Ønsket utdata**
x	-40	40	40	
y	40	40	-40	
figurrotasjon				225
midtavstand				57
bredde	20	20	20	20
høyde	20	20	20	20
farge	1	1	1	1
objektrotasjon	0	0	0	0

Tabell F.3: Variabler i figurrekke-oppgave 3.

Variabel	**Verdi 1**	**Verdi 2**	**Verdi 3**	**Ønsket utdata**
x	-40	-20	0	
y	40	20	0	
figurrotasjon				135
midtavstand				28
bredde	20	20	20	20
høyde	20	20	20	20
farge	0	1	0	1
objektrotasjon	0	0	0	0

Tabell F.4: Variabler i figurrekke-oppgave 4.

Variabel	**Verdi 1**	**Verdi 2**	**Verdi 3**	**Ønsket utdata**
x	-40	40	-40	
y	0	0	0	
figurrotasjon				90
midtavstand				40
bredde	20	20	20	20
høyde	20	20	20	20
farge	0	0	0	0
objektrotasjon	0	0	0	0

Tabell F.5: Variabler i figurrekke-oppgave 5.

Variabel	**Verdi 1**	**Verdi 2**	**Verdi 3**	**Ønsket utdata**
x	-40	0	40	
y	0	40	0	
figurrotasjon				180
midtavstand				40
bredde	20	10	20	10
høyde	10	20	10	20
farge	1	1	1	1
objektrotasjon	0	90	180	270

Tabell F.6: Resultater fra kjøring, oppgave 1.

Variabel	**Ønsket utdata**	**Utdata 1**	**Utdata 2**	**Utdata 3**	**Utdata 4**	**Utdata 5**
figurrotasjon	45	45	45	45	45	45
midtavstand	57	57	57	57	57	57
bredde	20	20	20	20	20	20
høyde	20	20	20	20	20	20
farge	0	0	0	0	0	0
objektrotasjon	0	0	0	0	0	0

Tabell F.7: Resultater fra kjøring, oppgave 2.

Variabel	**Ønsket utdata**	**Utdata 1**	**Utdata 2**	**Utdata 3**	**Utdata 4**	**Utdata 5**
figurrotasjon	225	225	225	225	225	225
midtavstand	57	57	57	57	57	57
bredde	20	20	20	20	20	20
høyde	20	20	20	20	20	20
farge	1	1	1	1	1	1
objektrotasjon	0	0	0	0	0	0

Tabell F.8: Resultater fra kjøring, oppgave 3.

Variabel	**Ønsket utdata**	**Utdata 1**	**Utdata 2**	**Utdata 3**	**Utdata 4**	**Utdata 5**
figurrotasjon	135	135	135	135	135	135
midtavstand	28	28	28	28	28	28
bredde	20	20	20	20	20	20
høyde	20	20	20	20	20	20
farge	1	1	1	1	1	1
objektrotasjon	0	0	0	0	0	0

Tabell F.9: Resultater fra kjøring, oppgave 4.

Variabel	**Ønsket utdata**	**Utdata 1**	**Utdata 2**	**Utdata 3**	**Utdata 4**	**Utdata 5**
figurrotasjon	90	90	90	90	90	90
midtavstand	40	40	40	40	40	40
bredde	20	20	20	20	20	20
høyde	20	20	20	20	20	20
farge	0	0	0	0	0	0
objektrotasjon	0	0	0	0	0	0

Tabell F.10: Resultater fra kjøring, oppgave 5.

Variabel	**Ønsket utdata**	**Utdata 1**	**Utdata 2**	**Utdata 3**	**Utdata 4**	**Utdata 5**
figurrotasjon	180	180	180	180	**63**	180
midtavstand	40	40	40	40	**89**	40
bredde	10	10	10	10	10	10
høyde	20	20	20	20	20	20
farge	1	1	1	1	1	1
objektrotasjon	270	270	270	270	270	270

Tillegg G

Estimat av en datamaskins score på en IQ-test

Dette vedlegget prøver å gi et estimat av hvor godt datamaskin vil score på en IQ-test beregnet på mennesker. Til grunn for beregningen av estimatet brukes NRKs IQ-test fra Test nasjonen 2004 [NRK04a].

IQ-testen består av totalt 68 oppgaver fordelt på fem kategorier, og hver kategori består av mellom en og fire oppgavetyper. Hver oppgavetype består av fire til seks oppgaver. Til hver av oppgavene er det gitt fire løsningsalternativer, noe som gjennomsnittlig ville gitt 17 av 68 poeng ved ren gjetting. For hver oppgavetype er det, i intervaller på 25 %, gitt et estimat for hvor stor sannsynlighet det er for at en datamaskin klarer å løse en oppgave av den gitte oppgavetypen.

Dersom vi antar at det er 50 % sannsynlighet for at datamaskinen får til en oppgave av gitt oppgavetype vil vi i estimatet gi denne oppgavetypen 3,75 av 6 mulige poeng. 3 av poengene stammer fra de 3 oppgavene vi tror datamaskinen klarer å løse i oppgavene, og 0,75 av poengene stammer fra at datamaskinen tipper tilfeldig på de tre siste oppgavene (3 * 0,25). Estimatet over hvor mange poeng vi antar datamaskinen vil få i hver oppgavetype er gjengitt i tabell G, mens tabell G, G og G begrunner hvorfor hver oppgavetype er gitt den sannsynligheten den er gitt.

Når vi har estimert hvor mange poeng datamaskinen ville fått på IQ-testen må vi slå opp i en tabell for å finne ut hvor mange IQ-poeng resultatet tilsvarer. Et lite utvalg av tabellen er gjengitt i tabell G, mens hele tabellen ble gitt i tv-programmet Test nasjonen 2004 [NRK04a].

Tabell G viser at vi kan anta at datamaskinen scorer 44 av 68 poeng på IQ-testen dersom forslagene til løsningsmetoder gjengitt i denne rapporten hadde blitt brukt. Ved å slå opp i tabell G ser vi at 44 poeng på testen tilsvarer, for alderstrinnet 20–34 år, 101 i IQ.

Tabell G.1: Estimat av poeng fordelt på oppgavetyper.

Språk				
Ord som ikke passer	50 %	5 oppgaver	3,125	
Sortere bokstaver	75 %	6 oppgaver	4,875	
Ordspråk	0 %	6 oppgaver	1,5	
Sum språk				*9,5*
Hukommelse				
Hukommelse	100 %	4 oppgaver	4	
Sum hukommelse				*4*
Logikk				
Figurrekker	50 %	6 oppgaver	3,75	
Tallrekker	100 %	5 oppgaver	5	
Figurmatriser	50 %	6 oppgaver	3,75	
Ordpar	25 %	6 oppgaver	2,625	
Sum logikk				*15,125*
Regning				
Tekstoppgaver	50 %	6 oppgaver	3,75	
Sum regning				*3,75*
Teknikk og romforståelse				
Teknikk	0 %	6 oppgaver	1,5	
Speilbilder	100 %	6 oppgaver	6	
3D-figurer	50 %	6 oppgaver	3,75	
Sum teknikk og romforståelse				*11,25*
Sum alle oppgaver				***43,625 ≈ 44***

Tabell G.2: Begrunnelse av oppgavetype-sannsynligheter, del 1.

Språk		
Ord som ikke passer	50 %	Hjelpemiddlene for å finne ut hvilke ord som passer sammen, og hvilke som ikke gjør det (kapittel 7.1.1) er bare middels gode. Disse må bli betydelig bedre for at datamaskinen skal score rett på alle oppgaver i kategorien.
Sortere bokstaver	75 %	Å sortere bokstaver (kapittel 7.1.2) er relativt enkelt, men hjelpemiddlene for å finne ut om et ord er i en gitt kategori har fortsatt en del å gå på.
Ordspråk	0 %	Det finnes per i dag ingen verktøy for å representere ordspråk (kapittel 7.1.3) slik at ordspråkets egentlige betydning kommer frem. Dermed er det også umulig å vite hvilke ordspråk som har lik betydning.
Hukommelse		
Hukommelse	100 %	Dersom man finner en god måte å representere det som skal huskes er hukommelsesoppgaver (kapittel 7.2.1) enkle for datamaskinen. Det som skal huskes er vanligvis enkelt å representere, og vi kan derfor anta at datamaskinen har en god sjanse til å løse slike oppgaver.

Tabell G.3: Begrunnelse av oppgavetype-sannsynligheter, del 2.

Logikk		
Figurrekker	50 %	Implementasjonen av et system for å løse figurrekker (kapittel 6) har vist at det er mulig å lage systemer for å løse en del av disse oppgavene. Variasjonene i rekkene er imidlertid så store at det kreves store ressurser for å lage programmer som kan løse disse alle oppgavene i kategorien.
Tallrekker	100 %	Resultatene fra kjøring av implementasjonen brukt for å løse tallrekker (kapittel 5) er overbevisende nok til at vi kan si at programmet klarer å løse de aller fleste tallrekkeoppgavene.
Figurmatriser	50 %	Oppgavene (kapittel 7.3.4) minner mye om figurrekker, og vi kan derfor anta at oppgavene er omtrent like vanskelige å løse som disse.
Ordpar	25 %	Hjelpemidlene for å finne hvordan ord står i sammenheng til hverandre (kapittel 7.3.5) er såpass dårlige at vi kun kan forvente oss at de vil klare å løse et fåtall av oppgavene.
Regning		
Tekstoppgaver	50 %	Det finnes et godt utvalg systemer som forstår naturlig språk. Problemet er at disse er begrenset til et domene, mens de tekstlige regneoppgavene (kapittel 7.4.1) kan strekke seg over et stort antall tenkelige domener. Det er likevel mulig å lage systemer som vil klare å løse en del av disse oppgavene.

Tabell G.4: Begrunnelse av oppgavetype-sannsynligheter, del 3.

Teknikk og romforståelse		
Teknikk	0 %	Teknikkoppgaver (kapittel 7.5.1) er svært varierte, og det er derfor svært vanskelig å lage en generell beskrivelse av hvordan slike oppgaver skal løses.
Speilbilder	100 %	Speilbildeoppgaver (kapittel 7.5.2) er relativt enkle. Ved å rotere bildene ser man om disse passer over hverandre. Når vi har funnet det av bildene som ikke passer over de andre har vi funnet løsningen, en oppgave som er enkel for datamaskinen.
3D-figurer	50 %	Det finnes allerede systemer for å lage 3D-figurer av 2D-figurer (kapittel 7.5.3). Svaralternativene kan dermed gjøres om til 3D og sammenlignes med 3D-figuren fra oppgaven.

Tabell G.5: Omregningstabell fra oppgavepoeng til IQ.

Poeng / Alder	**16–19**	**20–34**	**35–54**	**55–69**	**70+**
...	...	...	...	...	...
42	104	99	101	103	109
43	105	100	102	104	110
44	107	101	103	105	111
45	108	102	104	106	113
46	110	103	105	108	115
...	...	...	...	...	...

Bibliografi

[Abo05] About.com. Top 5 OCR Software for Windows. http://desktoppub.about.com/cs/win/tp/ocr_win.htm, 2005.

[Amb05] Tore Amble. BusTUC – A natural language bus route oracle. http://www.idi.ntnu.no/%7etagore/busstuc/, 2005.

[ASA05] Clue Norge ASA. Clue. http://www.clue-international.com/index_no.asp, 2005.

[AT&05] AT&T. The On-Line Encyclopedia of Integer Sequences. http://www.research.att.com/%7enjas/sequences/, 2005.

[Bar00] Robert Baruch. Groovy Java Genetic Programming. http://jgprog.sourceforge.net/, 2000.

[BBC05] BBC. Test The Nation. http://www.bbc.co.uk/testthenation/iq/, 2005.

[BNKF98] Wolfgang Banzhaf, Peter Nordin, Robert E. Keller, and Frank D. Francone. Genetic Programming — An Introduction. Morgan Kaufmann Publishers, Inc., 1998.

[Bra97] Jon S. Bratseth. Bustuc – A Natural Language Bus Traffic Information System. http://www.idi.ntnu.no/%7etagore/rapporter/bustuc.pdf, 1997.

[dSM94] Chantal de Séréville and Bernard Myers. Nøkkelen til suksess i alle IQ-tester. Bonniers Spesialblader og Bøker AS, 1994.

[ECJ05] ECJ. About Parameters and Parameter Files. http://cs.gmu.edu/%7eclab/projects/ecj/docs/parameters.html, 2005.

[Fer05] Jaime J. Fernandez. The GP Tutorial. http://www.geneticprogramming.com/Tutorial/, 2005.

[Fre05] Joshua Freedman. Eq directory. http://www.eq.org/, 2005.

[Gar99] Howard Gardner. Intelligence Reframed: Multiple intelligences. New York: Basic Books, 1999.

[Goo05] Google. Google. http://www.google.no/, 2005.

[Got97] Linda S. Gottfredson. Why g Matters: The Complexity of Everyday Life. http://mensa.dk/intelligens_i_100_aar.html, 1997.

[he05] "hansel" and "engelbrekt". Debattforum — Stemmer ikke helt. http://snakk.nrk.no/debatt/showthread.php?s=-941b56951e92adfacea62d4d96356757&threadid=31433, 2005.

[IC02] Karl Irikura and Jianquan Chen. Summary on Convert 2d to 3d in batch mode. http://www.ccl.net/cgi-bin/ccl/message.cgi?2002+07+21+005, 2002.

[Ils02] Steinar Ilstad. Generell psykologi. Tapir Akademisk Forlag, 2002.

[Inc05] Imagecom Incorporated. 2Dto3DCAD.com. http://www.aspire3d.com/Main.cfm, 2005.

[JM00] Daniel Jurafsky and James H. Martin. Speech and Language Processing — An Introduction to Natural Language Processing, Computational Linguistics, and Speech Recognition. Prentice Hall, 2000.

[Kar04] Benny Karpatschof. Faktoranalyse. http://www.psy.ku.dk/karpatschof/phd-stat-2004/Faktoranalyse.doc, 2004.

[Kei03] Maarten Keijzer. Scaled Symbolic Regression. Genetic Programming and Evolvable Machines, 5, 259–269, 2003.

[Kin05] Bill Kinnersley. The Language List. http://people.ku.edu/%7enkinners/-LangList/Extras/langlist.htm, 2005.

[Koz92] John R. Koza. Genetic Programming: On the Programming of Computer by Means of Natural Selection. The MIT Press, 1992.

[LPB+05] Sean Luke, Liviu Panait, Gabriel Balan, Zbigniew Skolicki, Jeff Bassett, Robert Hubley, and Alexander Chircop. ECJ. http://cs.gmu.edu/%7eeclab/projects/ecj/, 2005.

[Man04] Jan Manfeld. Intelligens i 100 år. http://mensa.dk/intelligens_i_100_aar.html, 2004.

[Nil98] Nils J. Nilsson. Artificial Iintelligence: A New Synthesis. Morgan Kaufmann Publishers, Inc., 1998.

[Nor05] Mensa Norge. Mensa Norge. http://www.mensa.no/, 2005.

[NRK03] NRK. Test nasjonen – Fasit 2003. http://www.nrk.no/programmer/tv/-test_nasjonen/3121955.html, 2003.

[NRK04a] NRK. NRK Nett-TV — Test nasjonen 2004. http://www7.nrk.no/nrkplayer/default.aspx?klipp_id=53162, 2004.

[NRK04b] NRK. Test nasjonen. http://www.nrk.no/programmer/tv/test_nasjonen/, 2004.

[NRK04c] NRK. Test nasjonen – Den store IQ-testen 2004. http://test.nrk.no/nrk/liveversion/iq_v3/flash.html, 2004.

[NTN05a] NTNU. BussTUC – Bussorakelet. http://www.idi.ntnu.no/%7etagore-/busstuc/, 2005.

[NTN05b] Det historisk-filosofiske fakultet NTNU. Tuc-prosjekter. http://mime.hf.ntnu.no:8080/hf/prosjekter/spraktek/prosjekter/tuc-prosjekter, 2005.

[oW05] University of Washington. KnowItAll. http://www.cs.washington.edu-/research/knowitall/, 2005.

[Set96] Ravi Sethi. Programming Languages — Concepts & Constructs. AT&T, 1996.

[SUN05] SUN. Java. http://java.sun.com/, 2005.

[Tho05] Simon Thoresen. Software archive 2004 – 2005. http://www.idi.ntnu.no/%7simonth/ntnu/pub/, 2005.

[Uni05] Princeton University. WordNet. http://wordnet.princeton.edu/, 2005.

[Wik05a] Wikipedia. C Plus Plus. http://en.wikipedia.org/wiki/C_Plus_Plus, 2005.

[Wik05b] Wikipedia. C programming language. http://en.wikipedia.org/wiki-/C_programming_language, 2005.

[Wik05c] Wikipedia. Java programming language. http://en.wikipedia.org/wiki/Java_programming_language, 2005.

[Wik05d] Wikipedia. Lisp programming language. http://en.wikipedia.org/wiki/Lisp_programming_language, 2005.

[Wik05e] Wikipedia. List of programming languages. http://en.wikipedia.org/wiki/List_of_programming_languages/, 2005.

[Wil05] Wilderdom. Wechsler Adult Intelligence Scale (WAIS). http://www.wilderdom.com/personality/intelligenceWAISWISC.html, 2005.

[YAd04] Martin Ystenes, Aftenposten, and diverse. Nettmøte med Martin Ystenes. http://tux1.aftenposten.no/nettprat/261104Ystenes/, 2004.