UNIVERSITETET I OSLO
Institutt for informatikk

Masteroppgave: Evolusjon for robotkontroll på Lego Mindstorms NXT

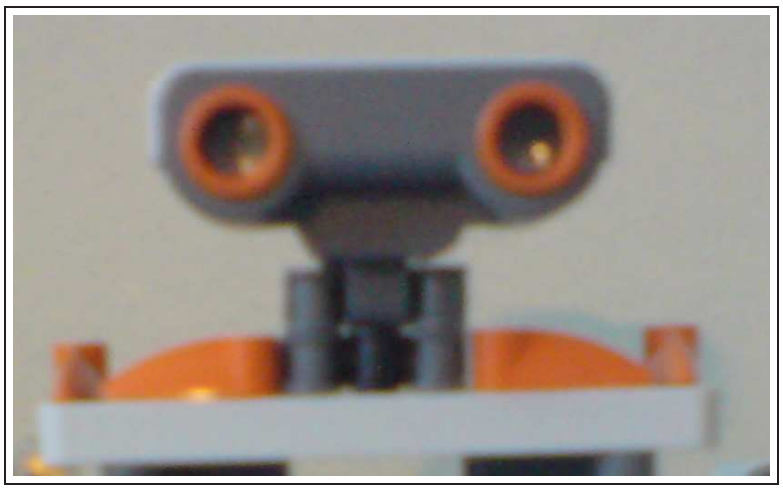
Christian A. Myrvold

August 2008

Sammendrag

Målet med denne oppgaven har vært å forsøke å gjenskape lignende studier med en nyere type robot som er mer tilgjengelig og billigere enn de mer vitenskapelige robotene. Et annet mål har også vært å forsøke dette med et nyere, og blant nye programmerere, mer populært språk, som Java. I løpet av oppgaven vil det også vises til begrensningene og fordelene med denne nye teknologien.

I denne oppgaven vil det bli sett på teknologier som evolusjonære algoritmer, nevrale nettverk, og kognitive kart. Disse vil bli testet på en Lego Mindstorms NXT-robot, i håp om at den kan være en god erstatter for de mer eksklusive robotene som typisk blir brukt i denne typen forsøk. Flere alternativer vil bli testet ut, og der det ikke ser ut til å kunne gjenskape resultatet i tidligere studier, vil det bli diskutert og testet ut nye løsninger.



Figur 1: Oscar

Innhold

Figurer

Tabeller

Forord

Denne masterrapporten beskriver mitt arbeide for Mastergrad i Logikk og kunstig intelligens for programmet IT: Språk, logikk og psykologi, i samarbeid med ROBIN-gruppen på Institutt for Informatikk, ved Universitetet i Oslo. Arbeidet av rapporten ble utført fra januar 2008 til august 2008.

Mange takk går til prof. Jim Tørressen for god veiledning og tålmodighet, og spesielt for å beholde troen på oppgaven når jeg begynte å miste min.

Jeg ønsker også å takke min familie og venner for forståelse og støtte under hele skriveprosessen. Takk også til medstudent Stig Bergestad for god idéveksling tidlig i prosessen, hvor mange av ideene bak denne oppgaven kom til overflaten for første gang.

Christian A. Myrvold

1 Introduksjon

I de siste 15 årene har evolusjon på robotkontroll blitt svært populært, og grensene for hva som kan gjøres utvides hvert år. I denne oppgaven utforskes emnet på flere områder. Det blir tatt i bruk den nyeste versjonen av Lego Mindstorms robotserien, NXT[1]. Eksperimenter med den forrige versjonen – RCX – har vært få i forhold til andre typer roboter, noe som gir rom til å utforske hva man har gjort med andre roboter og implementere det på dette nye robotverktøyet.

Samtidig vil det bli tatt i bruk Java som utviklingsspråk, og det vil bli sett på både fordeler og ulemper med dette. Java blir mer og mer populært å bruke siden det oftere blir studentens første møte med programmering, og det begynner nå å vokse ut en gruppe som ser på dette språket som sitt hovedprogrammeringspråk. Dermed kan det være interessant å se hva man kan få gjort med Java i en slik utviklingsprosess.

I kapittel 2 blir det sett på bakgrunnen for hele oppgaven, nemlig evolusjonstanken som inspirasjon for evolusjonære algoritmer, teknologien som finnes som samsvarer med de biologiske bestanddelene og relevante studier innen denne kategorien. I kapittel 3 ser jeg nærmere på teknologien bak Lego Mindstorms NXT-pakken og utviklingsverktøyet i Java. I kapittel 4 kommer en detaljert beskrivelse av det egenlagde programmet som tar i bruk de forskjellige teknologiene og tankegangene. I kapittel 5 vil det bli sett nærmere på forsøkene som ble foretatt under utviklingsprosessen, fra begynnelse til slutt. Til slutt vil det i kapittel 6 bli sett på mulig videreutvikling basert på denne oppgaven og i kapittel 7 konkluderes arbeidet som er gjort i denne oppgaven.

[1]Merk: verken dette eller 'RCX' er et akronym og forkortelsen har ingen betydning

2 Bakgrunn

I avsnitt 2.1 og 2.2 ser jeg på den teoriske og biologiske bakgrunnen til den evolusjonære fremgangsmåten. Så ser jeg på evolusjonære algoritmer i detalj i avsnitt 2.3, og så implementasjonen av dette i Java – JGAP – i avsnitt 2.4. I avsnitt 2.5 ser jeg litt på nevrale nettverk og implementasjonen som heter Joone. Så kommer et raskt innblikk i evolusjon av nevrale nettverk i avsnitt 2.6 og igjen implementasjonen av dette. Til slutt kommer et nærmere innblikk i litteraturen av evolusjon med roboter i avsnitt 2.7.

2.1 Darwinistisk evolusjon

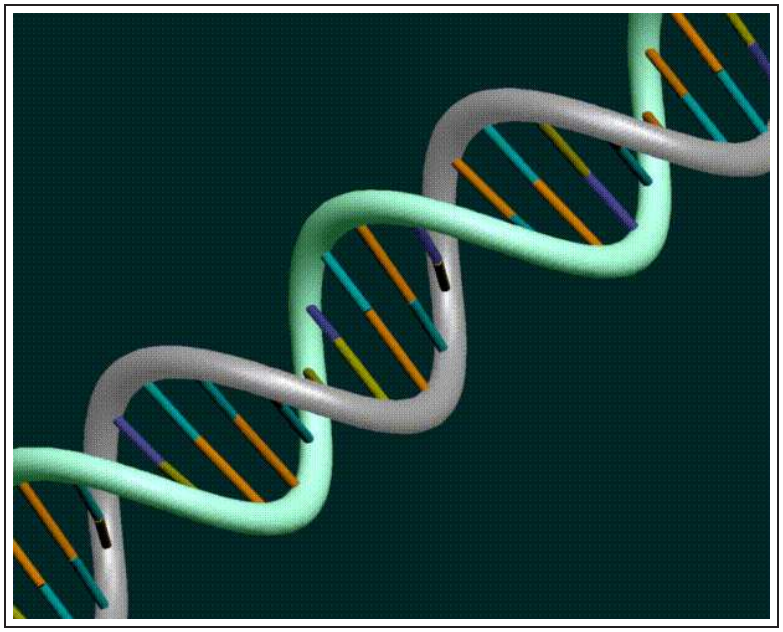
Tanken bak det å bruke Darwinistisk evolusjon til problemløsning kom så tidlig som på 1940-tallet (Turing), og ble først anvendt på en datamaskin av Bremermann i 1962. Men det var først på nittitallet at denne typen algoritmer virkelig kom i bruk, når datamaskinene begynte å bli kraftige nok til å kunne kjøre programmene hurtig nok til å kunne ha betydning.

Evolusjon i denne betydningen bygger veldig på Darwins teorier om ss-urvival of the fittest", altså at de individene som har tilpasset seg best vil kunne overleve lengre og ha større sannsynlighet for å formere seg. Dermed vil genene til dette individet – som har tilpasset seg godt til miljøet sitt – ha mulighet til å bli overført videre til neste generasjon. Egenskaper til individer som ikke overlever vil dermed kunne bli dempet eller forsvinne helt. Da ender man opp med en mengde individer som, selv om de er ulike, har tilpasset seg godt nok til å kunne overleve i det miljøet de lever i. Problemløsningsperspektivet kommer inn når vi ønsker å utvikle individer som har såpass gode tilpasningsevner at de kan overleve i mange forskjellige miljøer, og samtidig være lokalt tilpassete. Altså, de vil ikke være maksimalt tilpasset et problem, men være jevnt gode over mange forskjellige problemer. Mennesket er et godt eksempel på dette, som har utviklet seg til å kunne overleve nesten overalt på jorden, og har spredt seg over flere varierte miljøer.

2.2 Evolusjon

Hvis vi går dypere til bunns i organismen, kommer vi etter hvert til det genetiske nivået, selve byggeblokkene til organismen. Alle celler i et organisme innehar en kopi av det samme DNA'et (kromosomer) som alle andre celler, og kopierer dette videre ved celledeling. DNA er bygd opp av mange forskjellige gener, som igjen innehar den genetiske informasjonen som trengs for å bygge opp organismen. Genene bestemmer f.eks. hvilke proteiner som skal

produseres i hvilken del av organismen, og angir de forskjellige cellenes funksjon.



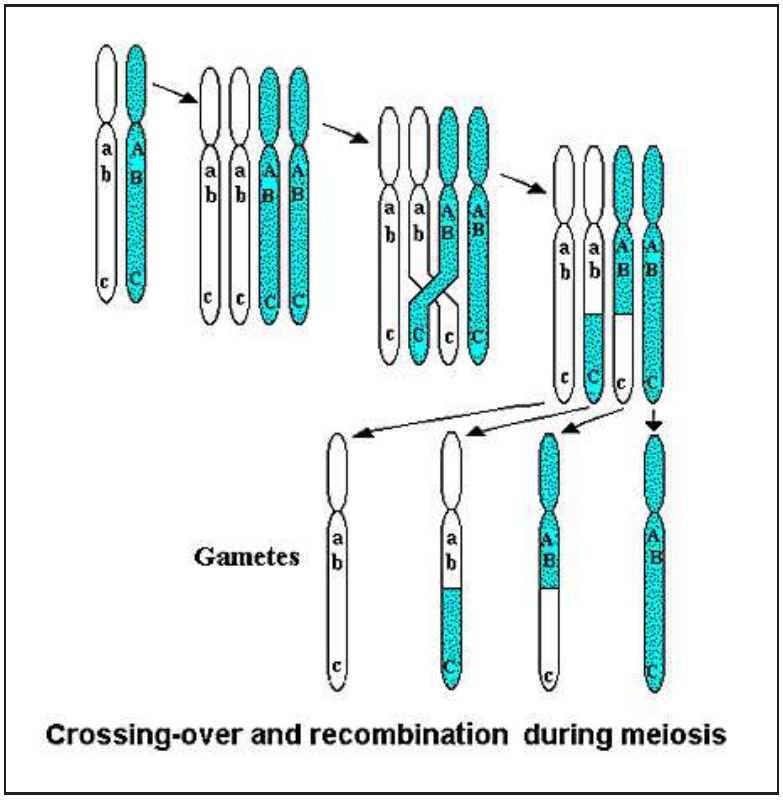
Figur 2: DNA-molekyl

Ved seksuell reproduksjon skjer en prosess som heter rekombinasjon (eller crossover; fig. 3). I en rekombinasjon brytes to DNA-strenger, bytter ut seksjoner med hverandre, og settes sammen igjen. Dermed dannes det nye kombinasjoner av kromosomer i det nye individet, og genetisk informasjon byttes ut.

Mutasjon kan også forekomme i celler, hvor DNA'et kan endres og celler kan begynne å produsere andre eller nye proteiner. Ofte skjer dette tilfeldig, og kan forekomme ved en feil ved celledeling, eller det kan forekomme ved stråling, kjemiske reaksjoner eller virusangrep. Naturlig utvalg vil avgjøre om mutasjonen er positiv og vil overleve i senere generasjoner. Dermed kan mutasjon gi variasjoner i genpoolen, og hvis endringen overlever vil det kunne føre til evolusjonen av nye arter.

2.3 Evolusjonære algoritmer

Tanken bak evolusjonære algoritmer (EA) [2] er å utnytte den darwinistiske evolusjonsteorien. Algoritmen jobber på det genetiske nivået og lager flere generasjoner av individer (mulige løsninger) som lærer og endrer seg i forhold

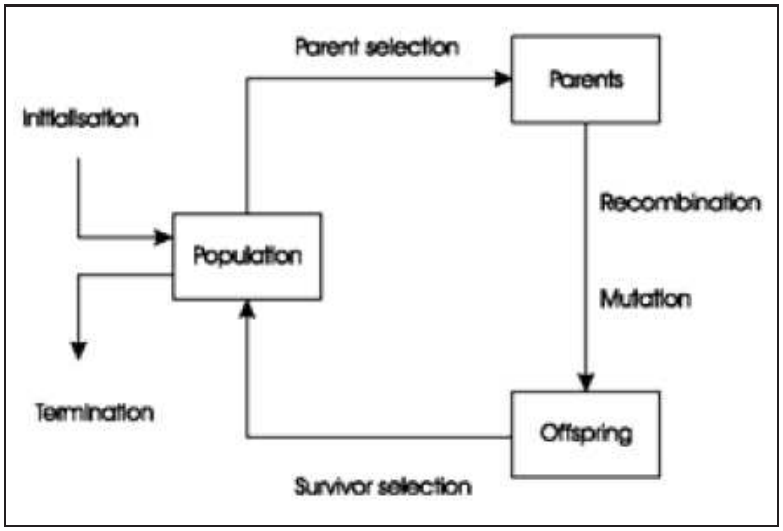


Figur 3: Rekombinasjon av gener

til miljøet (problemet som skal løses) de interagerer med (fig. 4). Mulige løsninger kombineres med andre, og løsninger som passer best til problemet har større sannsynlighet for å overleve til neste generasjon. Ofte brukes distinksjonen genotype og fenotype fra biologien for å skille mellom selve løsningen (fenotype) og kromosombeskrivelsen av løsningen (genotype). To løsninger kan ha samme fenotype – de ser like ut – men ha helt forskjellig genotype. Dette gjelder imidlertid ikke den andre veien.

2.3.1 Fitnessfunksjon

Fitnessfunksjonen brukes på alle individer for å si hvor bra hvert individ er i forhold til problemet som skal løses. Jo høyere fitness et individ har, jo høyere sannsynlighet er det for at det individet blir plukket ut til neste generasjon, eller vil kunne rekombineres, eller begge deler. Den sier egentlig bare noe om kvaliteten til individet i forhold til problemet. I dronningoppgaven [8] – plassér N dronninger på et N*N brett – vil dermed individene typisk bli evaluert i forhold til hvor mange dronninger som er plassert slik at de ikke



Figur 4: Overblikk over den evolusjonære algoritmen [2]

kolliderer med andre dronninger, og disse vil ha en høy fitness.

2.3.2 Populasjon

Populasjonen er mengden av individer, eller mulige løsninger. Typisk opprettes det først en tilfeldig populasjon. Over tid endres og utvikles populasjonen ved at individene blir endret eller byttet ut med nye individer. Vanligvis endres ikke størrelsen på populasjonen under evolusjonen. Ofte ser man også på differansen[2] til populasjonen. Dette er for å passe på at populasjonen ikke går mot altfor like individer[3], siden man ønsker en viss spredning av mulige løsninger. Perfekte løsninger er ikke alltid ønskelig, men heller mange gode – og forskjellige – løsninger.

2.3.3 Foreldreutvalg (Parent selection)

Ut i fra populasjonen velges det ut foreldre til nye individer basert på foreldreutvalg (parent selection). Disse blir valgt ut gjennom fitnessfunksjonen til individene, og bestemmer hvilke individer som får lov å bli foreldre for den neste generasjonen. Utvalget er egentlig bare en sannsynlighet for å kunne reprodusere, og ingen individer har noen garanti for å bli valgt ut til reproduksjon – uavhengig av hvilken fitnessverdi de har.

[2]eng.: diversity
[3]konvergering

Selve reproduksjonen kan skje på to måter i EA. Mutasjon skjer på ett individ og skaper ett nytt individ, hvor én eller flere egenskaper endres. Ofte er disse endringene små, men de er også helt tilfeldige og man vil ikke kunne vite på forhånd om mutasjonen er positiv eller negativ.

Rekombinasjon (eller crossover) skjer på to individer, og skaper like mange nye individer. På lik linje som mutasjon er det tilfeldig hvilke egenskaper som krysses fra de to foreldrene, og hvordan de krysses. Dette er det samme som seksuell reproduksjon i biologien, og dette prinsippet er blitt brukt av dyreavlere opp etter årene for å fremavle bestemte egenskaper. Det vil være stor sjanse for at de positive egenskapene til foreldrene vil arves av avkommet, og vil resultere i egenskaper som er bedre tilpasset miljøet avkommet skal leve videre i.

2.3.4 Overlevelsesutvalg (Survivor selection)

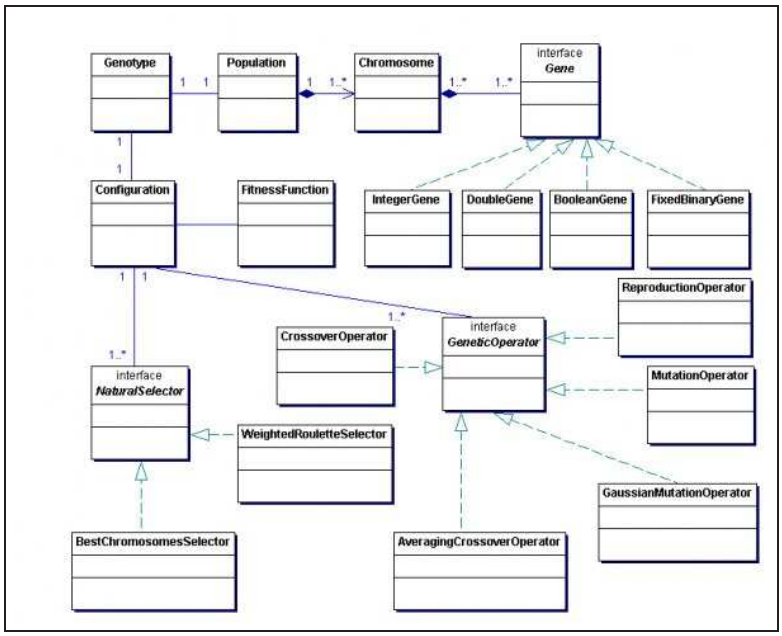
Etter at en ny generasjon er dannet, må vi innskrenke populasjonen til å holde seg innenfor den gitte populasjonsstørrelsen. Vi må altså gjøre et overlevelsesutvalg (survivor selection) for å avgjøre hvilke individer som får overleve til neste populasjon som igjen skal utvikles videre. Vanligvis velges det én av to strategier: enten bytter man ut alle gamle individer med avkommet (kan bare brukes hvis antall nye individer er lik antall gamle individer), eller så velger man ut de individene med best fitness uavhengig av alder.

2.3.5 Termineringsbetingelse

Til slutt må man bestemme når man skal avslutte søket, altså velge ut en passende termineringsbetingelse. Noen ganger er dette enkelt, f.eks. hvis det bare skal finnes én løsning. Men hvis vi leter etter en god nok løsning, må vi ta et valg. Man kan terminere søket etter en viss tid, typisk gitt i antall generasjoner, men da kan vi ikke vite noe om kvaliteten på løsningen. Eller vi kan terminere etter at vi har fått et individ med et forhåndsbestemt fitnessnivå. Problemet her vil være at vi ikke vet hvor lang tid det vil ta å komme frem til en slik løsning, eller i verste fall aldri nå den. Et annet problem her er at slike betingelser må settes av brukeren, noe som lett kan føre til en prøv-og-feilmentalitet.

2.4 JGAP

JGAP[4] er en Java-pakke for genetiske algoritmer og genetisk programmering som er lett å sette seg inn i (fig. 5). Pakken kommer også med flere test-eksempler, som knapsack-problemet [6] og traveling salesman-problemet [1].



Figur 5: Oversikt over de viktigste klassene i JGAP

2.4.1 Configuration-klassen

Configuration-klassen inneholder til enhver tid oppsettet av parametre og variable som er nødvendige for å utføre en genetisk algoritme – som fitnessfunksjon, naturlig selektor, genetiske operatorer osv.)

Objektet som dannes av denne klassen kan ikke endres etter at man lager en Genotype-objekt med konfigurasjonen. Ikke alle konfigurasjonsvalg er nødvendige, men jeg tar med de som blir brukt her.

2.4.2 NaturalSelector-klassen

Naturlige selektorer er ansvarlige for å velge ut et spesifisert antall kromosomer fra en populasjon, ved å bruke fitnessverdien som en guide. Vanligvis blir fitnessverdien brukt som en statistisk sannsynlighet for

[4]Java Genetic Algorithm Package

overlevelse, og ikke som den eneste bestemmende faktoren. Dermed vil man si at kromosomer med høy fitness har høyere sannsynlighet å overleve enn de med lav fitness, men vil aldri ha en garanti for dette.

Det finnes flere selektorer i JGAP, hvor jeg har brukt NaturalSelector-klassen. Det finnes også BestChromosomes-, StandardPost-, Threshold-, Tournament- og WeightedRouletteSelector.

2.4.3 GeneticOperator-interface

En genetisk operator representerer en operasjon på en populasjon av kromosomer under evolusjonsprosessen. Vanlige slike operasjoner er reproduksjon, crossover, og mutasjon. Dette interfacet inneholder bare én metode – operate() – som er ansvarlig for å utføre den genetiske operatoren på gjeldende populasjon av kromosomer.

2.4.4 FitnessFunction-klassen

Fitnessfunksjoner blir brukt for å bestemme hvor optimal en bestemt løsning er relativt til andre løsninger. Denne abstrakte klassen utvides av brukeren ved å overkjøre evaluate-metoden fra superklassen – brukerens klasse implementerer en egen evaluate(), som gjør det brukeren ønsker. Fitnessfunksjonen blir gitt et kromosom som den skal evaluere og bør returnere en positiv double som reflekterer fitnessverdien. Det er et krav at fitnessverdien skal være positiv, men ellers finnes det ingen restriksjoner på hvor stor den kan være; det er tross alt den relative forskjellen mellom kromosomenes fitnessverdier som avgjør sannsynligheten for et kromosoms overlevelse.

2.4.5 Genotype-klassen

Genotyper er populasjoner med fiksert mengde av kromosomobjekter. Når en instans av Genotypen blir evolvert, blir også kromosomene den inneholder evolvert. Som sagt før, må hver genotype ta i bruk den samme objektinstansieringen av Configuration, slik at alle genotyper har samme grunnlag.

2.4.6 Population-klassen

En populasjon er en liste av kromosomer som er inneholdt i genotypen. Denne klassen inneholder flere metoder for manipulasjon og innhenting av individuelle kromosomer.

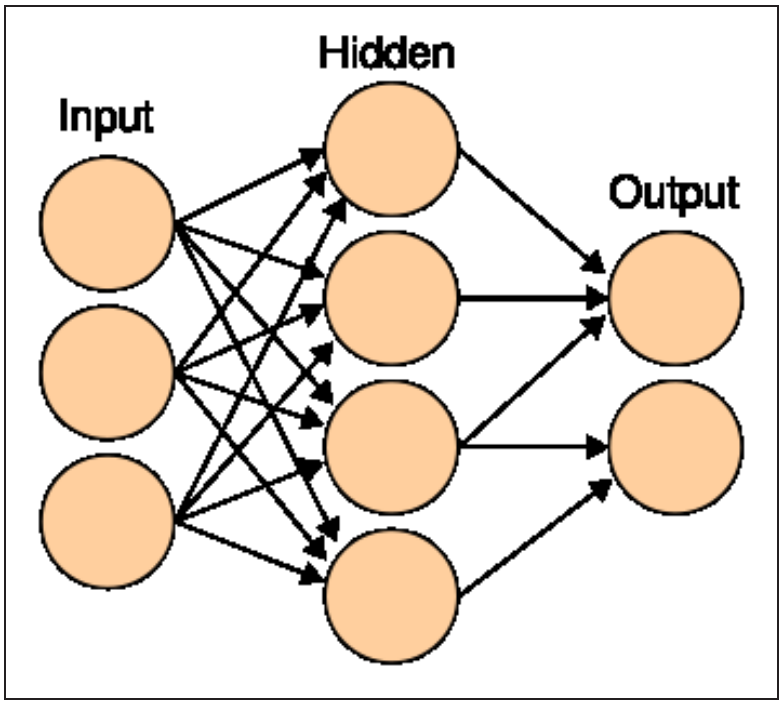
2.4.7 Chromosome-klassen

Kromosomer representerer potensielle løsninger og består av en genkolleksjon med fiksert lengde. Hvert gen representerer en diskret del av løsningen. Alle gener i en bestemt posisjon (locus) må dele den samme konkrete implementasjonen over kromosomer innen én enkel populasjon (genotype).

2.4.8 Gene-interface

Gener representerer enkeltkomponentene av en potensiell løsning (kromosomet). Interfacet eksisterer for at genimplementasjoner lett kan legges på[5], noe som gjør den mer fleksibel og brukbar for flere applikasjoner. Det er veldig viktig at implementasjoner av dette interfacet også implementerer en equals-metode. Uten denne implementasjonen kan det være at noen genetiske operatorer ikke vil fungere ordentlig.

2.5 Nevrale nettverk



Figur 6: Et eksempel på et lite nevralt nettverk

[5]f.eks. kromosomer av reelle tall, stringer, trær osv.

Tanken bak kunstige nevrale nettverk er inspirert av hvordan det biologiske nervesystemet prosesserer informasjon. Dette prosesseringssystemet inneholder et stort antall sammenkoblete nevroner – prosesseringselementer – som jobber sammen for å løse et spesifikt problem, som f.eks. mønstergjenkjenning. Løsninger fremkommer ved at koblingene som finnes mellom nevroner justeres i forhold til frekvensen (eller vekten/styrken) de blir brukt i gjeldende problem.

Den vanligste oppbygningen av et nevralt nettverk er en trelagsstruktur (fig. 6): et *inputlag* som har en forbindelse til et *skjult lag*, som igjen er koblet til et *outputlag*. Nettverket blir først trent opp med informasjon som blir sendt inn til nettverket via inputnodene, slik at vektene kan endre i forhold til informasjonen. Vektene mellom input- og det skjulte laget bestemmer når nodene i det skjulte laget er aktive, og ved å endre disse vektene kan de endre hva de representerer. Hva outputnodene gjør avhenger av hvordan de skjulte nodene oppfører seg og endrer vektene mellom det skjulte og outputlaget. Senere tester man nettverket med liknende informasjon for å se om det har lært seg å gjenkjenne inputen eller gi riktig respons/output. Et slikt nettverk kaller man typisk et adaptivt nettverk, og er det vanligste å bruke.

Innen kunstig intelligens-feltet har nevrale nettverk også blitt brukt innen talegjenkjenning, bildeanalyse og tilpassende kontroll, for eksempel for autonome roboter.

2.5.1 Joone

Joone[6] er et nevralt nettverk-rammeverk skrevet i Java. Den er bygd opp av hovedkjernen, en GUI-editor og et distribuert treningsmiljø og kan utvides gjennom å skrive nye moduler som implementerer egne algoritmer eller arkitekturer gjennom basiskomponentene.

Det er et gratis rammeverk for å lage, trene og teste nevrale nettverk. Tanken var å lage et kraftig miljø både for entusiaster og profesjonelle brukere, basert på de nyeste Java-teknologiene. Joones nevrale nettverk kan bygges på en lokal maskin, trenes på et distribuert system og kjøre på hvilken som helst datamaskin.

Tanken var å bruke noen av hovedklassene i Joone for å lage nevrale nettverk, noe som i og for seg fungerer aldeles utmerket. Programvaren har allerede kommet til versjon 2.0.0, som vil gjøre det enklere å bruke API'en – som kan virke litt komplekst første gang man ser den – og ytelsen er økt med 50%, med høyere sikkerhet og skalerbarhet.

[6]Java Object Oriented Neural Engine

2.6 Evolusjon av nevrale nettverk

For at man skal kunne bruke nevrale nettverk, må man bestemme passende grenseverdier for alle vektene og antall noder i lagene. Slike endringer av verdier vil ofte føre til en prøv-og-feiltaktikk, noe som kan bli svært tidskrevende å gjøre manuelt. Derimot passer det ypperlig for en evolusjonær fremgangsmåte. Det finnes flere muligheter for valg av populasjon, men et vanlig alternativ er å ha en populasjon av vekter, som blir justert og testet i forhold til den evolusjonære algoritmen. De vektene som gir gode resultater vil dermed ha bedre muligheter til å overleve til neste generasjon.

2.6.1 JooneGap

JooneGap er ment å være en bro mellom de to bibliotekene JGAP og Joone, slik at man kan kjøre evolusjon på nevrale nettverk i Java uten å måtte gjøre noe egen koding for dette. Dessverre ble siste versjon (0.4, beta) sluppet i juli i 2005, og er aldri blitt oppdatert senere i forhold til nye versjoner av JGAP.

2.7 Litteratur

2.7.1 Evolusjon med roboter

I 1998 gjorde Jean-Arcady Meyer et studie av artikler som har gjort nettopp eksperimenter innebærende evolusjon av nevral kontroll med roboter [5]. Dette studiet gir en god oversikt over de forskjellige fremgangsmåtene forskere har tatt, samt hva slags roboter de tok i bruk. Den vanligste roboten var Khepera (se figur 7), som ble brukt av nærmere halvparten, med forskjeller i atferd (fra unngåelse av hinder til områderengjøring), type nevral kontroller (f.eks. perceptron), type genotype (f.eks. vektene på det nevrale nettverket) og om de brukte læring. Det Meyer fant ut i 1998 er at enkelte har hatt suksess i å evolvere mer kognitive arkitekturer med enkel motorikk. Det store spørsmålet var bare hvor godt det ville skalere opp mot mer komplekse problemer.

2.7.2 Khepera

Som sagt var roboten Khepera veldig populær, og har vært en inspirasjonskilde for roboten bygget og brukt i denne oppgaven. Den vanlige varianten[7] er sirkulær i formen, med en diameter på 55mm, en høyde på 30mm, og en vekt på 70g. Den har to hjul for fremdrift og to teflonballer for balanse. Den enkle formen gjør at Khepera lett skal kunne navigere rundt enhver type hinder

[7]Merk at dataene fra denne varianten er fra 1994

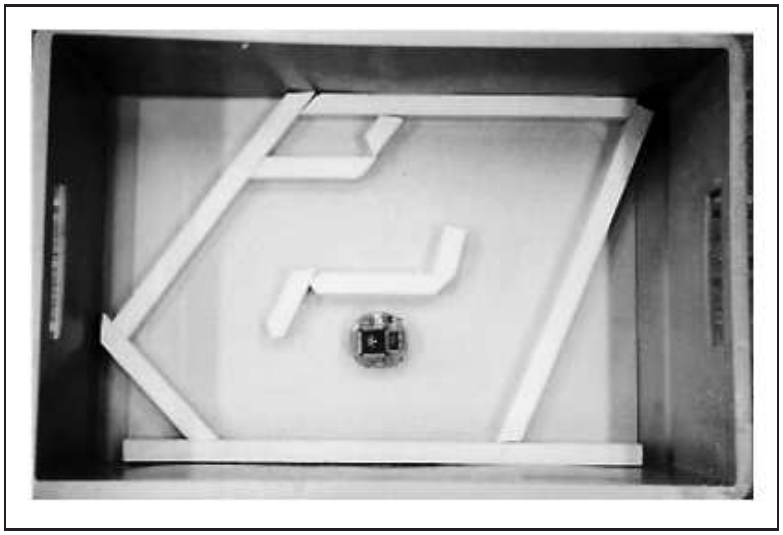
Figur 7: Roboten Khepera

og hjørner. Den er utstyrt med seks infrarødsensorer foran og to bak. Den inneholder også en kontroller på 256Kbytes RAM og 512 Kbytes ROM som administrerer alle input-output-rutinene og kan kommunisere via en seriell port med en vertsdatamaskin. Den lille størrelsen på roboten gjør det også enkelt å bruke den i laboratorieeksperimenter. Det finnes også en variant med en gripearm foran, og roboten er designet for at man skal kunne gjøre egne modifikasjoner.

2.7.3 Floreanos Khepera-eksperiment

Floreano [3] tok i bruk Kheperaroboten i 1994 for å vise fremkomsten av karakteristikker til autonome agenter. For å gjøre dette gjorde de evolusjon på nevrale nettverk på robotkontrollen og viser at de beste individene fremviser en fullstendig utnyttelse av ikke-lineære forbindelser som er mer effektive enn agenter designet av mennesker.

Det som først og fremst inspirerte meg med denne artikkelen var den enkle målsettingen og utførelsen. Roboten skulle utvikle strategier for å unngå hindre og samtidig øke farten. Eksperimentet viste fremkomsten av fenomener som selvregulering av farten og kjøring fremover. Dette virket som mulig å replikere med roboten jeg hadde laget, selv om det var en viss forskjell i antall og type sensorer som kunne tas i bruk.



Figur 8: Experimentmiljøet for Khepera

2.7.4 Simulert bilkjøring

I Julian Togelius' artikkel [7] fra 2006 beskrev han evolusjonen av kontrollere for å kjøre en simulert radiostyrt bil rundt en bane. Eksperimentet ble brukt til å sammenligne forskjellige kontrollerarkitekturer, og finne ut hvilken som passer best for å kunne utvikle en god kjøreatferd.

Selv om studiet i seg selv var interessant, var det enda mer interessant på grunn av én liten detalj: de utdypet hvordan outputen fra det nevrale nettverket ble tolket som handlinger brukt av bilen. Hvis aktiveringen var mindre enn -0.3 ble tolket som baklengs, mer enn 0.3 som fremover og alt i mellom ikke ga noen motorhandling. Dette ønsket jeg å teste uten å bruke nevrale nettverk, men bare med evolusjon og kromosomene som aktiveringsverdien.

2.7.5 Kognitive kart

Kognitive kart innenfor kunstig intelligens handler om ønsket å kopiere menneskets evne til å gjenskape et omtrentlig kart av omgivelsene og kunne navigere etter dette kartet. For mobile roboter er dette essensielt for å kunne navigere og utforske et fremmed miljø. I simuleringer blir det enda viktigere å ha et velfungerende kognitivt kart å navigere etter. Typisk har programmet minimalt med reelle data å jobbe med i simuleringen og det mangler en fysisk robot som kan oppdatere kartet i reell tid ved bruk av sensoravlesninger.

Alle artikler rundt dette emnet har typisk hatt behov for å kunne lage svært detaljerte kart, slik at roboten vet nøyaktig hvordan omgivelsene ser ut. Men for å lage slike avanserte kart trenger man også instrumenter som er avanserte nok avanserte nok – som infrarøde sensorer – ellers vil ikke roboten kunne navigere etter det. Et litt ekstremt tilfelle var et studie gjort av Andreas Kurz [4] i 1996, hvor han utstyrte roboten sin med 24 ultrasoniske sensorer. Dette var selvfølgelig umulig for meg å gjenskape – særlig når det ser ut som om roboten er på størrelse med et menneske. Men jeg ønsket å se hva man kunne gjøre med to rimelig enkle sensorer og skape et enkelt kognitivt kart ut av dette. Samtidig ville det være interessant å se hvor bra et slikt kart er for navigasjon.

3 Lego Mindstorm teknologi

Her vil jeg vise til forskjellige teknologier og bibliotek jeg har har brukt for både robot og i selve implementasjonen. I avsnitt 3.1 går jeg i detaljer på selve roboten som er laget med Lego Mindstorms NXT-settet, og i avsnitt 3.2 beskrives biblioteket som ble brukt for å kunne kommunisere med og styre roboten.

3.1 Lego Mindstorms NXT

I 1998 kom Lego ut med sin helt nye serie, Lego Mindstorms. Denne serien var ment som både en ny adspredelse for entusiaster og et opplæringsverktøy for studenter, og ble først tatt i bruk av MIT (hent referanse fra wiki) for å lettvint kunne lage roboter uten store kunnskaper om robotikk og mikroelektronikk. Første versjon – RCX – kom med en programmerbar brikke, tre sensorer, Lego-brikker og Lego Technic-brikker.

Den neste versjonen – NXT – kom i 2006 og er den versjonen som er blitt brukt i denne oppgaven. Ikke bare er programmeringsbrikken kraftig oppgradert, den inneholder også én ekstra sensor, den ultrasoniske sensoren. Videre vil jeg se nærmere på de forskjellige delene.

Figur 9: Rex-roboten som Lego bruker til å promotere Mindstorms-pakken

3.1.1 Programmeringsbrikken (NXT-brikken)

Dette er hovedbrikken (fig. 10) for enhver Lego Mindstorms robot, ikke bare fordi det er her programmeringen skjer, men også siden all bygging vil skje rundt denne brikken. Den har innganger til fire sensorer og tre motorer, og kobles til gjennom RJ12 kabler. Brikken har også en LCD-skjerm på 100x64 piksler og fire knapper for navigasjon av brukergrensesnittet. Brikken har også en høytaler for avspilling av svært enkle lydfiler. Brikken får strøm fra 6 AA-batterier.

Figur 10: NXT-brikken

Tekniske spesifikasjoner:

- 32-bit ARM7 mikrokontroller
- 256 KB flashminne og 64 KB RAM
- 8-bit AVR mikrokontroller
- 4 KB flashminne og 512 Bytes RAM
- Bluetooth class II trådløs forbindelse, som kan brukes for å overføre programmer til NXT-brikken eller for å fjernstyre roboten med f.eks. mobiltelefon
- 1 USB 2.0-port (12 MB/s)
- 4 input-porter (for sensorer)
- 3 utput-porter (for motorer)
- 100x64-piksel LCD-display
- Høytaler - 8 kHz lydkvalitet. Lydkanal med 8-bit oppløsning og maks 16 KHz lydrate.

3.1.2 Trykksensor

Denne sensoren (fig. 11) brukes ikke i denne oppgaven, men kunne ha blitt brukt for å detektere objekter som roboten kjører på. Sensoren har da to tilstander, av og på, avhengig av om knappenpå sensoren blir trykt inn eller ikke.

Figur 11: Trykksensoren

3.1.3 Lydsensor

Lydsensoren (fig. 12) brukes heller ikke i denne oppgaven, først og fremst fordi jeg ikke så et relevant bruksområde for den. Sensoren måler lyder i desibel (dB) og justert desibel (dBA). Måling av justert desibel registrerer lyder som det menneskelige øret kan høre, mens desibelmålingen ikke skiller mellom disse og tar med lyder som både kan være for høye eller lave for det menneskelige øret å høre.

Figur 12: Lydsensoren

3.1.4 Lyssensor

Med lyssensoren (fig. 13) kan roboten skille mellom lys og mørke, og kan lese av lysstyrken i nærheten eller lysintensiteten til fargete overflater. Farger vil for sensoren komme frem som gråtoner. Noe man må passe på derimot, er at sensoren har et eget lys, typisk for å måle reflektert lys fra objekter, og at

dette kan ødelegge for enkelte målinger, som f.eks. lysintensitet i nærheten. Dette lyset er derfor slått av i denne oppgaven.

Figur 13: Lyssensoren

3.1.5 Ultrasonisk sensor

Den ultrasoniske sensoren (fig. 14) gir roboten muligheten til å se og oppdage objekter. Den kan også måle avstand til nærmeste objekt - opptil 255 cm (+- 3 cm) - og oppdage bevegelse. For å regne ut distanse sender den ut en ultrasonisk puls (ping), og hvis det er et objekt i pulsretningen, blir deler av eller hele pulsen reflektert tilbake til sensoren som et ekko. Distansen regnes ut gjennom tiden det tar for dette ekkoet til å registreres i sensoren.

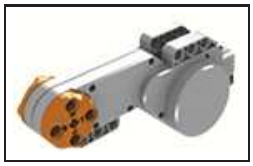
Figur 14: Den ultrasoniske sensoren

3.1.6 Servomotorene

Noe av de aller viktigste delene av roboten er ansvarlig for fremdrift. Hver motor (fig. 15) har også en rotasjonssensor innebygd, som gjør at styringen av motorene er nøyaktige. Denne sensoren måler rotasjonen til motoren i grader, hvor 360 grader er én hel omdreining.

3.1.7 Kompass-sensoren

Denne sensoren (fig. 16) inkluderes ikke i Mindstorm NXT-pakken, og må kjøpes separat. Den måler jordas magnetfelt og regner hvilken retning



Figur 15: Én av motorene

sensoren er rettet mot. Kompasset har innebygd kalibrering for å redusere den magnetiske forstyrrelsen fra andre kilder. Dessverre viste det seg at NXJ ikke støttet denne brikken, og dermed ble den til slutt heller ikke brukt.

Figur 16: Kompass-sensoren; kjøpes separat

3.2 leJOS NXJ / iCommand

leJOS[8] er et open-source høynivåspråk basert på Java, og leJOS NXJ har mange klasser for kommunikasjon med de forskjellige sensorene og motorene i Lego Mindstorms NXT. Det er to måter å bruke dette biblioteket på: man kan enten laste opp hele programmet sitt over til NXT-brikken slik at roboten blir helt autonom og uavhengig av en PC, eller så kan man bruke iCommand for å kjøre programmet på en lokal PC og kommunisere med roboten over Bluetooth.

iCommand[9] er altså en Java-pakke for å styre NXT-brikken med en Bluetooth-forbindelse, og er det som brukes her. Den bruker NXJ-koden for å motta kommandoer trådløst fra et Java-program på en PC. Utviklingen av iCommand er avhengig av utviklingen av NXJ, og vil dermed ligge litt bak NXJ-biblioteket. Praktisk sett betyr dette at en del klasser og metoder som er videreutviklet i NXJ enten er utdatert eller ikke eksisterer i iCommand.

[8] webadresse: http://lejos.sourceforge.net/index.php

[9] versjon 0.7 brukt her

3.2.1 Motor-klassen

Her kan man lage tre instanser for de tre motorene man kan koble til NXT-brikken. Kontroll av motorene gjøres med forward, backward og stop-metodene. Man kan regulere motorenes fart med metoden setSpeed. Hastighet blir her regnet ut i antall grader per sekund. Noe man må merke seg her er at motorenes hastighet avhenger av hvor mye strøm man har igjen på batteriene.

3.2.2 UltrasonicSensor-klassen

Med denne klassen lager man en instans for kontroll av den ultrasoniske sensoren. Noen nyttige metoder:

- setMetric(): her kan man sette distansemålingen til å bli målt i centimeter, i forhold til tommer som er standarden.
- getDistance(): returnerer distansen fra sensoren til nærmeste objekt som returnerer pulsen fra sensoren.

3.2.3 LightSensor-klassen

Klassen for kontroll av lyssensoren. Her har man mulighet for å kalibrere for lave og høye verdier. Noen nyttige metoder:

- getLight({Percent/Value}): få lysverdien fra sensoren enten i prosent eller den normaliserte verdien.
- passivate()/activate: disse skrur respektivt av og på LED-lyset til sensoren. Her blir lyset skrudd av siden sensoren vil kunne få lysrefleksjonen sendt tilbake, og det kan gi feilaktig avlesning av de reelle verdiene i miljøet.

3.2.4 CompassSensor-klassen

Klasse for å kontroll av kompassensoren. Noen nyttige metoder:

- getDegrees(): leser av kompasset og returner retningen i grader.
- getDegreesCartesian(): kompassavlesning øker klokkvis fra 0 til 360 grader, men kartesiske koordinatsystemer øker mot klokka.
- startCalibration(): kalibrerer kompasset, kan være greit å gjøre ved instansiering for å ha gjort det.

3.2.5 NXTCommand

Klassen som muliggjør fjernkontroll til en NXT-brikke via Bluetooth. Veldig enkel i bruk, man bruker bare open-metoden for å opprette en kontakt ved å bruke iCommand, setVerify(true) for å låse kontakten, og close() for å lukke kontakten. Hvis man ikke lukker kontakten på denne måten, må man slå NXT-brikken av og på mellom hver gang.

3.3 Bluetooth

Bluetooth er en kommunikasjonsprotokoll med lav kraftforbruk, typisk med en lav rekkevidde. Denne teknologien gjør at to enheter med Bluetooth innebygd kan kommunisere med hverandre når de er i rekkevidde. Enhetene bruker et radiokommunikasjonssystem, så de trenger ikke en gang være i samme rom, så lenge forbindelsen er sterk nok.

NXT-brikken har innebygd klasse-II Bluetooth, som gir den en rekkevidde på 10 meter.

4 Implementasjon

I dette kapittelet vil jeg utdype de delene av oppgaven som er laget selv. I avsnitt 4.1 vises selve roboten frem, som er laget av egen inspirasjon med litt hjelp av byggeinstruksjoner fra Lego. I avsnitt 4.2 vil selve programmet bli gått gjennom, punktvis så langt det lar seg gjøre. Avsnitt 4.3 er en klasseoversikt med forklaringer på klassene, valgene av dem og hva de har ansvar for.

4.1 Roboten Oscar

Figur 17: Første obligatoriske Lego-robot

Etter en del eksperimentering med forskjellige fasonger på roboten, endte jeg til slutt på det endelige produktet. Uten noe særlig erfaring innenfor robotbygging – utenom å leke med Lego i barndommen – ble det mye prøving og feiling. Dette er nok ikke en optimal robotfasong, men det var det beste man kunne gjøre med de byggeklossene man begynner med og erfaringen jeg har med bygging av roboter. Det var flere detaljer jeg ønsket å få frem, først og fremst:

- at sensorene kunne plasseres enten foran eller bak roboten, at man lett kunne bytte ut sensorer med hverandre, og at de ikke kom i veien for hverandre eller fremdriften,

(a) Foran

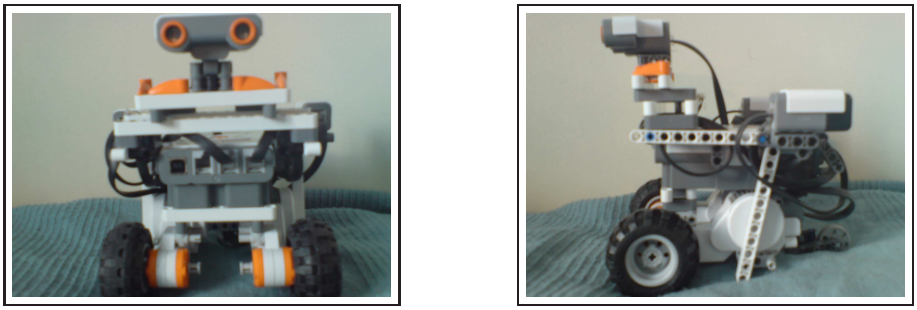
(b) Fra siden

Figur 18: Første ordentlige robot

- at jeg hadde en enkel fremdrift, som kunne snu seg raskt og uten problemer,
- at kablene – som er i forskjellige lengder – ikke kom i veien for hjulene og sensorene (se fig. 17 som eksempel),
- at roboten ikke ble altfor stor, noe som ikke er så lett når byggeklossene (ikke minst NXT-brikken) er såpass store i forhold til vanlige robotdeler,
- og at NXT-brikken var lett å få ut, uten å måtte bryte opp hele roboten. Dette var viktig med tanke på bytte av batteri, som holder til under NXT-brikken.

Det endelige produktet – som jeg har kalt Oscar – er en robot med to hjul for fremdrift som ligger foran, et balansehjul bak, den ultrasoniske sensoren foran – hvor man ville forventet at hodet ville være plassert – og muligheten for to sensorer bak. Det er også mulig å plassere to sensorer foran ved å legge til armlignende brikker under "hodet". NXT-brikken er enkel å demontere, man trenger bare å fjerne én brikke på hver side som holder NXT-brikken fast til resten av kroppen. Kablene ble surret rundt diverse deler av kroppen", hvor det var store nok hull til å få gjennom kablene.

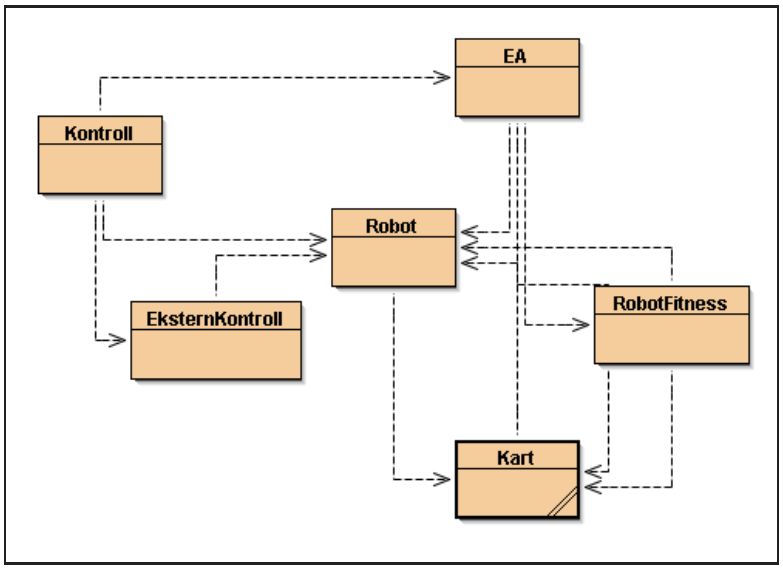
Et problem med utformingen er delen som holder hodet på plass – halsen. Dessverre var det få alternativer igjen for å få festet halsen til resten av kroppen, og løsningen ble at halsen sitter litt løst på. Dette betyr at hodet har en tendens å bevege seg nedover, enten over tid eller ved kraftigere rykk. For å løse dette har jeg forsøkt å feste halsen fast med både teip og ståltråd, men de holder bare i kortere tid.



(a) Foran (b) Fra siden

Figur 19: Siste versjon av Oscar

4.2 Programstruktur



Figur 20: Programstrukturen

Når programmet begynner å kjøre, vil den først forsøke å kontakte roboten via Bluetooth. Klarer den ikke det, vil programmet avslutte med en feilmelding. Hvis alt går bra, vil man få to valgmuligheter:

1. Kjøre hovedprogrammet, som automatisk setter i gang med evolusjonen og autonom styring av roboten.

2. Kjøre et testprogram, som tillater at brukeren selv styrer roboten. Her vil man ha mulighet til å kjøre roboten fremover, bakover og til siden. I tillegg kan man printe ut viktige opplysninger som batteristyrke, sensorinput og antall rotasjoner motorene har gjort siden siste printing.

Selve hovedprogrammet setter så i gang evolusjonsalgoritmen, og går så inn i en bestemt rekkefølge av handlinger:

- Programmet lager en instans av klassen EA, som gjør alt klart for den evolusjonære algoritmen; initialiserer populasjonen og setter opp konfigurasjonen for evolusjonen i JGAP – som kromosomselektor, crossoveroperator, populasjonsstørrelse. Alle stegene i evolusjonen blir håndtert av JGAP-programmet, mens dette programmet bestemmer hvordan kromosomene ser ut og hvordan fitnessfunksjonen skal tolke kromosomene og avgjøre hva som er bra. Her vil også kartet bli initialisert med hva avlesningen av sensorene tilsier.

- Når alt er klart setter evolusjonen i gang, og går gjennom disse punktene:

  - Les av sensorene, og hold av plassen i løsningsmatrisen som de to sensoravlesningene vil tilsi.
  - Hvis det allerede finnes en populasjon på denne plassen blir det beste kromosomet i stående populasjon utført. Hvis fitnessen er bedre enn den var før, blir gjeldende kromosom oppdatert med den nye fitnessen; men hvis fitnessen er verre, blir denne populasjonen evolvert X generasjoner i håp om å forbedre populasjonen.
  - Hvis det ikke finnes en populasjon, lages det en ny populasjon som evolveres i X generasjoner, og den beste løsningen utføres. Deretter lagres populasjonen i løsningsmatrisen for mulig senere bruk.
  - Til slutt oppdateres kartet med de nye dataene, og sendt til robotklassen – grunnen til dette er at robotobjektet ligger i Fitnessobjektet, og dermed har fitnessfunksjonen også tilgang til det samme kartet. Dessuten virket det ganske logisk at roboten har tilgang til kartet.

- **Fitnessberegning:** Når JGAP begynner med evolusjonsprosessen, blir objektet for fitnessberegningen sendt med, og fitnessfunksjonen går gjennom disse stegene:

- Løpende kromosom blir sendt som argument til fitnessfunksjonsmetoden evaluate() i fitnessobjektet. Dette kromosomet blir så brutt ned til gennivået, og bestanddelene blir analysert.
- De to reelle tallene blir satt til sine respektive motorparametre som deretter blir gjort om til faktiske omdreininger av motorene; eller motorenes fart.
- Så blir robotens retning hentet ut, og beregnet på nytt i forhold til motorenes fart. Dette er egentlig bare ett stort logisk uttrykk som endrer retningen i forhold til hvilke motorer som beveger seg, hvor mye de beveger seg, og i hvilken retning de beveger seg. Ut fra dette blir retningen enten uendret eller endret med +- 1 eller 2.
- Motorenes fart blir forenklet for simuleringen på kartet; hvis det er nok fremdrift vil den gå fremover og returnere tallet 1; hvis det er nok fremdrift bakover vil den returnere tallet 2; ellers returnerer den tallet 0, for ingen fremdrift.
- Så blir disse dataene brukt til å simulere løsningen på kartet. Denne vil i forhold til den nye retningen og fremdriften endre på robotens posisjon i kartet og beregne en fitnessverdi ved å se på robotens nye posisjon i forhold til hindre. Ingen hindre foran eller bak gir positive verdier, ellers blir det dårligere verdier. Mer om kartet i neste punkt.
- Så blir neste fitnessverdi beregnet ut fra motorenes nye fart. Denne sier egentlig bare at hvis det er nok fart i begge hjulene blir fitnessverdien høyere.

- **Kartsimulering:** Ved beregning av fitnessverdi er det nødvendig å simulere løsningen istedenfor å faktisk utføre den på roboten, og dette gjøres i kartobjektet.
  - Kartet blir initialisert som et 4x4 boolsk rutenett hvor roboten automatisk blir plassert på koordinatene x=2, y=2. Retning blir initialisert med tallet 1, med en rekkevidde på 1-8. Et hinder blir satt på kartet med 'true', ellers er det 'false'.
  - simuler() er hovedmetoden i denne klassen, og vil også returnere fitnessverdien for hindre som er i veien for roboten. I forhold til retningen som ble beregnet i fitnessobjektet, vil den sjekke spesifikke koordinater på kartet hindre og sette eventuelle hindre foran og bak på nytt.

  - Hvis roboten simulerer en bevegelse eller ser et hinder utenfor kartet, vil kartet rehashes. Dette innebærer at kartet blir doblet og alle hindre vil bli satt på kartet igjen med nye koordinater. Det samme gjelder robotens koordinater.

- **Robotens bevegelse:** Alle bevegelser som gjøres av roboten vil bli håndtert av robotobjektet, og vil bare få kommando for dette av evousjonsobjektet. Dette objektet håndterer også sensormålinger.

  - kjør-metoden håndterer selve oversettelsen av de reelle tallene den får fra den evolusjonære algoritmen til faktisk bevegelse av motorene. Dette gjøres på en enkel måte; verdier mellom 0.00-0.05 og 0.50-0.55 betyr stopp. Verdier mellom 0.05-0.50 betyr at den skal kjøre bakover. Alle verdier over 0.55 vil resultere i at roboten kjører fremover. Dermed har motorene like stor sannsynlighet til å kjøre fremover som bakover. Samtidig er det en 10% sjanse for at motoren står stille.
  - Når farten på motorene er satt vil metoden kjøre motorene samtidig i nøyaktig 300 millisekunder, for så å stoppe dem. Valget på 300ms ble beregnet ved at det var så lang tid det tok roboten å kjøre 13cm med full motorakselerasjon, som er like langt som roboten er og like langt som én rute på kartet. Dette var for at roboten aldri skulle klare å kjøre mer enn dette på én handling.

På grunn av roboten, kartet, og de mange interaksjonene dem i mellom ble programmet en del større enn først antatt. Vanligvis trenger ikke programmer som tar i bruk evolusjonære algoritmer å være så veldig store, men når man både må simulere handlingene – uavhengig av hvor enkelt det er blitt gjort her – og utføre dem på en robot, blir det fort en del ting å holde styr på; og dermed blir også programmet automatisk større.

4.3 Klasseoversikt

Dette er en rask gjennomgang av klassene i det egenlagde programmet, og er ment som et tillegg til programstrukturen (fig. 20 i avsnitt 4.2) og dermed ikke utfyllende på egen hånd.

4.3.1 Robot-klassen

Denne klassen er forbindelsen mellom leJOS-pakken, selve roboten og resten av programmet. Her skal all kontroll av roboten ligge, samt all annen bruk av leJOS-klasser som Motor-, LightSensor- og UltraSonicSensor-klassen. Den

har bl.a. metoder for å tolke kromosomverdier samt for omjustering av verdier fra roboten, når det føles naturlig.

4.3.2 EA-klassen

Denne klassen er bindeleddet mellom JGAP-pakken og resten av programmet. Den har egentlig bare én oppgave; lag populasjoner, utfør evolusjonen og overfør kommandoer til roboten i forhold til hva evolusjonen tilsier. Den overfører også verdier fra roboten til kartobjektet slik at kartet kan oppdateres, og sender kartet tilbake til robotobjektet for bruk i fitnessfunksjonen.

Fitnessverdi av gamle løsninger blir noen ganger beregnet og endret i denne klassen, siden gamle løsninger noen ganger endrer fitnessverdi i forhold til nye situasjoner. Da må kromosomet endres, og dette gjøres her.

4.3.3 Robotfitness-klassen

Dette er en subklasse av den abstrakte FitnessFunction-klassen i JGAP. Her avgjøres hva som er bra og hva som er mindre bra løsninger. Per i dag gjøres dette ved å si at motorene må ha en fart på minst 50% av maks fart for å registrere en økt fitnessverdi, samt at ikke det skal registreres noen hindre foran sensorene. Maksimal fitnessverdi er 4.0. All fitnessberegning gjøres i denne klassen gjennom kartobjektet, som simulerer løsninger i forhold til hva som frem til nå er registrert på kartet. Kartobjektet innhentes gjennom robotobjektet, som blir sendt med ved instansiering av klassen.

4.3.4 Kart-klassen

Kartet er en enkel representasjon av miljøet som roboten beveger seg i. Kartet er en todimensjonal boolsk dobbelarray, hvor verdien *true* merker kartkoordinatene som plassering av et hinder. Robotkoordinatene holdes separat fra kartet som x- og y-verdier.

Mye av denne klassen er metoder for manipulering av kartet. Det finnes en setObstacle-metode for å registrere hindre på kartet riktig i forhold til robotens retning, hvilken sensorverdi som tilsier at det finnes et hinder foran eller bak, og hvor langt unna objektet er. En annen metode simulerer robotens bevegelser og regner ut en fitnessverdi i forhold til hva man vet eksisterer i miljøet ut fra kartet.

5 Eksperimenter

Utover oppgavetiden er det blitt gjennomført noen tester, typisk for å teste ut moduler, algoritmer eller robotbrikker. Her kommer jeg til å gå i detalj hva som ble testet, under hvilke forhold, og eventuelle resultater, enten positive eller negative. Under tidlige forsøk (avsnitt 5.1) ser jeg på de forskjellige testene jeg gjennomførte, og som førte til den progresjonen jeg har hatt med hele masteroppgaven, og hvilke endringer jeg foretok i forhold til resultatene fra disse testene. Under senere forsøk (avsnitt 5.2) ser jeg på det endelige forsøket som er resultatet av foregående tester.

5.1 Tidlige forsøk

5.1.1 Test av evolusjon

Denne testen ble gjort på første versjon av Oscar (fig. 18), og var en enkel test for å se om evolusjonspakken fungerte. Det ble først dannet en populasjon på 20 individer, hver med en kromosomstørrelse på 6 gener (3 handlinger (se under "Robotens bevegelser" i avsnitt 4.2 for detaljer), og utførte 50 generasjoner. Dette tok cirka 30 minutter.

Testbanen var en raskt sammensatt kvadratisk bane med fire vegger og ett stort hinder i midten av banen. Dermed ble det dannet en vei rundt hvor man kommer tilbake til utgangspunktet. Håpet var å se en klar antydning til at roboten etter hvert ville unngå hindre, eller i det hele tatt evolvere frem løsninger som gjorde at roboten ikke kjørte rett mot hindre. Det var ingen forventninger at roboten ville komme med intelligente løsninger, bare at man kunne se forskjell fra begynnelsen av testen mot slutten av testen. Det var altså en ren kvalitativ test i forhold til kantitativ, altså resultatet ville først og fremst være skjønnsmessig. Siden jeg måtte ha en fitnessfunksjon, brukte jeg en provisorisk funksjon som ga høye verdier på positive løsninger og lavere verdier på negative løsninger. Men siden det ble vanskelig å se på disse dataene for å avgjøre om roboten faktisk viste tegn på endringer i kjøremønster, bestemte jeg meg for å konsentrere meg på hva jeg kunne se.

Roboten hadde på denne stund bare koblet på to sensorer, den ultrasoniske sensoren og lyssensoren, og så langt satt begge på fronten av roboten. I begynnelsen var kjøremønsteret – som forventet – helt tilfeldig, noe som var ganske naturlig siden løsningene i begynnelsen var helt tilfeldige. Men etter hvert – særlig mot slutten – kunne man se en bestemt endring i kjøremønsteret. Roboten kjørte færre og færre ganger fremover, og valgte heller

å kjøre bakover. Dette var ganske naturlig siden begge sensorene var satt på fremsiden av roboten, og enhver bevegelse bakover førte til at hindrene som roboten visste om beveget seg vekk fra den.

Dessverre førte dette ofte til at roboten kjørte seg fast ved å kjøre bakover inn i vegger, og fortsatte å kjøre bakover. Et annet problem her var at roboten kunne kjøre seg fast på en slik måte at ett av hjulene hang over bakken, og dermed ikke fikk noe friksjon. Dette førte til at det hjulet fremviste en veldig bra fitness, siden den kunne yte maksimalt uten at noe holdt hjulet tilbake. Dette var svært vanskelig å gjøre noe med, siden jeg ikke kunne måle vekten på hjulet – mindre vekt ville overført til hjulet hvis den ikke hadde kontakt med bakken – og dermed ikke kunne gi negativ fitness for denne forekomsten.

Et tredje problem var termineringsbetingelsen. Her ble det gjort enkelt ved å avslutte etter 50 generasjoner, men senere ville det kanskje ikke være bra nok å bare avslutte etter et gitt antall generasjoner, og heller ha en annen termineringsbetingelse. To andre muligheter ble vurdert; avslutte etter etter et gitt antall runder rundt banen, og avslutte etter at en bestemt fitness ble oversteget. På den første muligheten lå vanskeligheten i å avgjøre automatisk at roboten faktisk hadde kjørt rundt hele banen, og ikke bare kjøre frem og tilbake. Med den andre muligheten var det vanskelig å vurdere hva som var en bra fitness-score, og hva den fitness-scoren faktisk sa. Det ble også vurdert å bare la brukeren terminere evolusjonen når han eller hun så at roboten hadde kjørt en runde. Dette var ikke et ønskelig alternativ – det vanlige med evolusjonære algoritmer er å la den gå og komme tilbake om noen timer – men ville vært bra nok som et siste alternativ.

5.1.2 Test av evolusjon av nevrale nettverk

Denne testen kom dessverre aldri langt, først og fremst på grunn av tekniske problemer med programvare for evolusjon av nevrale nettverk, JooneGap (avsnitt 2.6.1). Etter å ha fulgt noen enkle eksempler av slik evolusjon, forsøkte jeg å sette i gang noe eget. Men raskt begynte feilmeldingene å dukke opp, og jeg merket at for å få dette til å fungere måtte det større endringer til. Løsningen rundt dette ble å enten kode hele overgangen mellom JGAP og Joone selv – og håpe at dette fungerte bra nok – eller gå inn i koden til JooneGap og oppdatere alt – og håpe at dette ikke ødela strukturen nok til at programmet ikke lenger fungerer slik det er tenkt.

JooneGap har heller ingen valgmuligheter for utvidende nevrale nettverk og bruk av egen fitnessfunksjon med bruk av egen software og/eller hardware.

Kjøringsnr.	Venstre motor	Høyre motor
1	250	208
2	238	207
3	278	223
4	377	241
5	341	235

Tabell 1: Data av motorrotasjon på 1 sekund

Når jeg koblet fitnessfunksjonen min – med tilkobling mot roboten – ble metodene for roboten aldri kjørt. Alt i alt følte jeg at det ble så mange problemer at arbeidet med å oppdatere og fikse problemene virket større enn resten av oppgaven, og jeg valgte å ikke lenger gjøre evolusjon av nevrale nettverk, og konsentrerte meg heller om ren evolusjon.

5.1.3 Test av backtracking

Denne testen kom som et resultat av problemene med den forrige testen, og hvordan jeg kunne overføre noe tankegangen til ren evolusjon. Et alternativ jeg gjorde et forsøk på her var å bruke backtracking", altså reversere den forrige handlingen slik at roboten skulle ende opp på samme sted før neste handling ble iverksatt. Tanken var å teste ut forskjellige løsninger, bruke den beste løsningen, kjøre en ny generasjon i evolusjonen, teste ut de neste løsningene osv. Dessverre viste denne testen at det var flere problemer å ta hensyn til.

Det største problemet var med motorene. Det virket ikke som om motorene helt gjorde det de ble bedt om, så jeg forsøkte å bare teste ut motorene. Dette gjorde jeg ved å sette roboten på hodet, slik at hjulene stod rett opp i lufta uten at friksjon fra gulvet kunne ha noen påvirkning. Ved hver test ble samme input gitt til metodene for styring av motorene – samme fart og samme tidsspenn – og jeg målte antall rotasjoner fra motorene. Her viste det seg at den samme inputen ikke vil gi den samme outputen. I tabell 1 følger data fra fem kjøringer, alle med samme input, og output fra begge motorene.

I et forsøk på å komme rundt dette, forsøkte jeg bruke noen andre metoder fra NXJ-biblioteket. Jeg hadde muligheten til å angi til motorene antall rotasjoner den skulle utføre, og dermed kunne jeg komme utenom den litt mer upresise metoden med fart og tidsspenn på handlingen. Dessverre var denne metoden enda mer tilfeldig, og viste seg å være ubrukelig. I tabell 2

Kjøringsnr.	Venstre motor	Høyre motor
1	-496	-249
2	669	339
3	-265	-7
4	482	229
5	-109	13

Tabell 2: Data av motorrotasjon på 300

følger data fra fem kjøringer med denne metoden, hvor hver motor blir bedt om å rotere med en verdi på 300.

Alt dette førte til at jeg valgte å gå vekk fra denne fremgangsmåten, og forsøke å finne en annen mulighet.

5.1.4 Test med kart og kompass

Den neste tanken jeg hadde var å bruke kompasset man kunne få til roboten – siste versjon av Oscar (fig. 19) for å kunne få en presis retningsavlesning, og bruke dette sammen med et automatisk generert kart. Kartet skulle brukes til simulering av potensielle løsninger i forhold til å utføre enhver potensiell løsning på roboten. Kartet i seg selv så ut til å fungere bra nok, men så fikk jeg problemet med kompasset. Selv om NXJ-biblioteket hadde metoder for avlesning av kompassbrikker, viste deg seg at jeg ikke hadde riktig kompassbrikke. Alt jeg kunne få ut var helt statiske verdier – det negative tallet '-291' – uansett hvilken vei roboten vendte seg.

Dette var en feil det var vanskelig å gjøre noe med, bortsett fra å kjøpe en ny kompassbrikke som forhåpentligvis var riktig, og jeg valgte å fjerne kompasset og gjøre det med en egen metode.

5.2 Senere forsøk

Etter å ha implementert en erstatning for kompasset (se under "Fitnessberegning" i avsnitt 4.2), og konfigurert grenseverdiene for denne metoden etter observerte data, var det på tide med det siste forsøket. Dessverre kunne man på forhånd se at motorene ville gi forskjellige utslag på resultatet her, og grenseverdiene ble svært provisoriske og lite nøyaktige. På grunn av tekniske problemer med JGAP var det ikke mulig å velge akkurat den naturlige selektoren jeg ønsket å bruke – vektet roulette-selektor – siden den bare ga en NullPointerException. Til slutt endte valget på beste-kromosomselektoren, som velger ut de N beste kromosomene og går videre med dem. Til å begynne

med startet jeg med en populasjon på 10 individer – hver med en kromosomstørrelse på 2 – som gikk over 20 generasjon Dette kunne utvides ettersom man ønsket å eksperimentere med antall individer, kromosomstørrelse og generasjoner, men som en første test var dette bra nok.

Jeg startet med et åpent miljø med mange veier å kjøre, og hvor roboten var stilt rett mot et hinder. Dermed måtte roboten allerede i første steg unngå et hinder ved enten å svinge eller kjøre bakover. Dette fungerte bra, og roboten svingte til siden – tre av fem ganger til venstre, to av fem til høyre, og aldri bakover. Dette virket lovende, siden handlingen å kjøre bakover i denne situasjonen fortsatt ville ha latt hinderet stå foran. Ved å svinge ble hinderet borte fra sensorene, noe roboten anser som en positiv løsning. Men siden roboten var i et åpent miljø, var det i de neste handlingene som roboten gjorde vanskelig å se noe intelligens. Det ble mer tilfeldige handlinger, siden den ikke hadde noen umiddelbare hindre å passe seg for. Kartet var også vanskelig å få ordentlig greie på, og det var ikke helt klart når man så på det hvilke hindre på kartet som samsvarte med det virkelige miljøet. Et annet problem som dukket opp i denne testen var nøyaktigheten på lyssensoren, som noen ganger virket å registrere hindre bak selv om det ikke var noe i umiddelbar nærhet. Dette er et resultat av to faktorer; eksperimentmiljøet mangler en lyskilde som er konstant og ikke-vekslende – naturlig lys utenfra endrer seg i løpet av dagen og kan ha stor påvirkning på lyssensoren; lyssensoren ble kalibrert i én type lyssetting, med grenseverdier for hva den skal tolke som umiddelbar nærhet til hinder bakover (e.g. én rute bakover i kartet), nærhet til hinder bak (to ruter bakover i kartet) og ingen hinder bak. Men med et vekslende naturlig lys vil disse grenseverdiene ikke lenger ha samme betydning, og lyssensoren kan "lures" til å tro at det er et hinder bak når det ikke er det. En løsning på dette ville vært å implementere en kalibreringsmetode, hvor man før en test kan gjenta lystestene for å avgjøre grenseverdiene på nytt med gjeldende lyskilde. Dette har jeg ikke implementert, først og fremst på grunn av tid, men også fordi det ville vært tidskrevende og lite smidig å måtte rekalibrere lyssensoren før hver test – særlig med tanke på at det ved første (og gjeldende) kalibrering ble brukt gjennomsnitt på flere tester. Da hadde det vært bedre å hatt en ordentlig lyskilde – som et lys direkte over testmiljøet – men det hadde jeg da ikke mulighet til.

Deretter ble det gjort forsøk med et mer lukket miljø – 70cm x 120cm – med ett stort hinder – 30cm x 25cm – midt i banen. Dette ga banen en nærmest sirkulær form, noe som gjorde det lettere å se mulig progresjon i handlingene til roboten. Roboten ble satt på én av langsidene i banen, uten hindre foran eller bak, og ble så satt i gang. Denne gangen ble resultatet

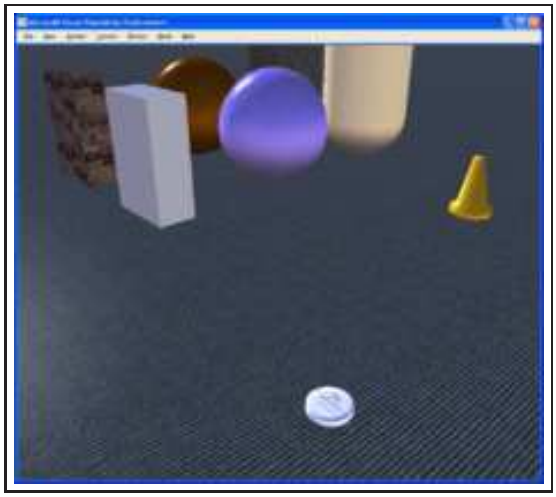
mer entydig, men ikke nødvendigvis mer positivt. Flere ganger rygget den inn i hindre på grunn av lyssensoren, og noen ganger kjørte den også på skrå inn i hindre. Sistnevnte problem dukket opp på grunn av den unøyaktige retningsmetoden jeg hadde implementert, noe som ble klart når man så på kartet i forhold til banen. Simuleringen ga beskjed til roboten at handlingen ville gi et positivt resultat, mens virkeligheten var at handlingen ville føre til at roboten kjører på et hinder med den ene siden. Et annet resultat var også at kartet oftere ble feil i forhold til virkeligheten, og når simuleringen mente det var greit å svinge i forhold til kartet, møtte roboten noen ganger veggen enn en åpen vei. Dette var helt klart på grunn av både den unøyaktige retningsmetoden og motorenes sporadiske output. Selv om den ble kjørt i flere generasjoner – oppimot 100 generasjoner – ble ikke resultatet noe bedre.

6 Videre arbeid

Innenfor dette området er det store muligheter til å gjøre noe som fungerer, og videre arbeid vil nok fokusere på evolusjon av nevrale nettverk (se avsnitt 2.6). Man må finne bibliotek som fungerer, og i Java er det mye som fortsatt er i beta-versjon, eller som ikke lenger støttes eller videreutvikles, og dermed er det stor sannsynlighet for at det er noe som ikke lenger samsvarer med annen teknologi. Det er flere ting som skal kunne fungere sammen, og da kan det fort bli til at noe bryter sammen eller at man må utvikle noe selv. En annen mulighet er å finne et bibliotek som er laget for nettopp evolusjon av nevrale nettverk, enten i Java eller et annet språk.

6.1 Microsoft Robotics Studio (MRS)

I løpet av dette semesteret ble jeg invitert til presentasjon og introduksjon til utviklingsmiljø for roboter kalt Microsoft Robotics Studio, blant annet også for Lego Mindstorms NXT. Dette er et Windows-basert miljø for robotkontroll og simulering, og vil kunne fjerne en god del egen utvikling av metoder som allerede eksisterer men som man ellers ikke har tilgang til. Simuleringen er visuell (se fig. 21), ser veldig profesjonell ut og inkluderer simuleringsmiljøer som fotball, sumo eller labyrint, hvor sistnevnte nok er den mest interessante delen.



Figur 21: Simuleringsmiljøet i MRS

Dessverre kom denne introduksjonen litt sent for min del, og jeg fikk ikke tid til å se nærmere på dette og muligheten til å bruke programmet i denne

oppgaven. Men av det jeg fikk sett virket dette ganske lovende, og støtter programmeringsspråk som C, Visual Basic .NET, JScript og IronPython. Gjennom å bruke et slikt program vil man nok kunne unngå en del av problemene jeg kom borti under utviklingen, som å samkjøre robotkontroll med simulering – og kart – uten å måtte bruke ekstra tid på å utvikle det selv. I tillegg er mange av komponentene i MRS standard, som PhysX-simulatoren fra ageia. Det store spørsmålet blir i så fall hvordan dette kan kobles opp mot bibliotek med evolusjonære algoritmer og nevrale nettverk.

7 Konklusjoner

Når man leser artikler rundt evolusjon av nevrale nettverk, ser man at det er stort potensiale for å gjøre noe stort innenfor dette feltet. Men det artiklene ikke forteller om er ofte det store problemet rundt denne teknologien. Det ligger veldig mye erfaring og tidligere tester og forsøk bak dem, og uten denne kunnskapen blir det svært vanskelig å vite nøyaktig hva de har gjort. Personlig har jeg hatt et lite kurs rundt evolusjon, og ingen kurs rundt nevrale nettverk. Hver for seg er de enkle å sette seg inn i, og med enkle eksempler forstår man raskt hvordan de fungerer, selv om det finnes mange flere variasjoner og muligheter for å gjøre det samme på ulike måter. Problemet oppstår når artiklene ikke uttaler det forfatterne vet altfor godt, f.eks. hvordan overføres signalene i det nevrale nettverket til robotens bevegelser. Dette gjør at artikler ikke egner seg til å lære seg teknologien. Legg til at de aller færreste tar med fitnessfunksjonen de har tatt i bruk, og man får i tillegg problemer med å korrekt gjenskape forsøkene de har utført i artiklene.

Valg av bibliotek vil typisk avhenge av språkpreferanse. Mitt valg av Java som programmeringsspråk var først og fremst basert på hvilket språk jeg kan best, og hvor jeg kjenner til språkets særegenheter. Dette reduserte valg av bibliotek betraktelig, særlig siden de fleste bibliotek bruker språk som er mer innarbeidet, som C og C++. Selv om jeg har hatt kontakt med disse språkene, følte jeg at det var bedre å velge et språk jeg kunne og ikke måtte bruke ekstra tid på å sette meg inn i detaljer jeg ikke har vært bort i. Selv om bibliotekene JGAP og Joone i seg selv virker veldig bra, hadde det vært fint med flere alternativer, særlig med tanke på det forlatte prosjektet JooneGap. Interessen for å gjøre evolusjon på nevrale nettverk i Java er til stede, men det vil nok bli for mye arbeid å ha en tredjeparts utvikler som ikke bare skal kode koblingen mellom disse bibliotekene, men også oppdatere sin egen kode hver gang én av disse blir oppdatert. Det vil nok være lettere å kunne bruke et bibliotek som gjør alt selv, og ikke er avhengig av andre utviklere. Dessverre har det ikke vært lett å finne et slikt bibliotek, særlig hvis det skal være gratis og open-source, slik at man ikke blir bundet av utvikleren i hvordan man tar i bruk biblioteket. Så her vil man nok enten måtte velge et mer omgjengelig programmeringsspråk eller håpe at det kommer bedre løsninger i Java.

Med tanke på Lego Mindstorms NXT-roboten er det vanskelig å finne ut hvor problemet ligger. Denne pakken fra Lego er tross alt rimelig ny – den kom ut høsten 2006 – og det er mulig at ikke alle tredjeparts utviklere utenom Lego har fått finjustert sine bibliotek for kontroll av NXT-brikken

og dens sensorer, særlig tredjepartssensorene. Når det gjelder problematikken med motorene, kan problemet ligge flere steder; det kan være at bibliotekene ikke har helt korrekt kode for presis kontroll, det kan være at NXT-brikken feiltolker inputen den får fra biblioteket, eller så kan det være at motorene er feilkalibrert – selv om det virker rart at alle motorene er feilkalibrert. Uansett er det et problem det er vanskelig å komme unna. En mulig løsning her kan være MRS-programvaren – hvis problemet faktisk ligger i biblioteket – men igjen så har jeg ikke fått mulighet til å teste dette. Et annet problem jeg fort ble gjort oppmerksom på, var mangelen på antall sensorer som kunne festes til NXT-brikken – bare 4 sensorer – i forhold til andre lignende roboter, som Khepera (avsnitt 2.7.2). Problemet ligger ikke så mye i at det begrenser hva roboten kan gjøre, men heller at det blir vanskelig å rettferdig sammenligne resultater.

For å avrunde så har det åpenbart vært en problematisk utviklingsperiode. Det virker nesten som om Murphys lov har inntredt, hvor nærmest alt som kan gå galt har gått galt. Dette har ikke bare gått hardt innpå hva jeg faktisk har fått gjort med oppgaven, men også på min egen entusiasme og engasjement. Når test etter test feiler blir det etter hvert vanskelig å holde inspirasjonen oppe, og jeg merker at oppgaven blir mer og mer fokusert på hvilke problemer som har oppstått og hvordan jeg har forsøkt å komme rundt dem. Når det endelige produktet har kommet frem gjennom prøving og feiling istedenfor å holde meg til en etablert teori, blir resultatet heller ikke så vitenskapelig sunt. Å gjenskape testene her vil bli svært vanskelig med tanke på lokale variasjoner – jf. motorene i avsnitt 5.1.3 – og en metode for å finne riktig retning i forhold til kartet som på sitt beste er utilregnelig (avsnitt 5.2). Det hjalp nok heller ikke at metodikken ikke var basert på noen overordnet teori, men heller et resultat av en prøv-og-feil-mentalitet. På en annen side er det som sagt vanskelig å finne kilder som gir detaljert nok informasjon til at man kan forsøke å gjenskape eksperimentene som dokumenteres i artiklene.

I retrospekt er det også lettere å se hva man kunne gjort annerledes, og kanskje hvor det gikk galt. Det er selvfølgelig lett å legge all skyld på de forskjellige programmene som ikke fungerte helt som man ønsket, men samtidig skal jeg ikke påstå at jeg håndterte alt riktig. Istedenfor å gi opp fremgangsmåten og implementasjonen jeg fant kunne jeg ha tatt oppgaven på egen kappe og gjort noe implementasjon selv, som f.eks. i JooneGap. Når problemene først begynte å inntreffe følte jeg at det var for dårlig tid til å gjøre dette, og at det muligens ville tatt over innholdet i oppgaven. Men ved skrivende stund er det lettere å se at jeg kunne hatt tid til å gjøre et ordentlig forsøk i å redde levningene. Samtidig burde jeg startet med et

noe enklere startpunkt med mindre kompleksitet, istedenfor å hoppe uti noe jeg kanskje ikke var klar for bare fordi noe å virket å fungere (jf. JGAP i avsnitt 2.4) i tidlige tester. Personlig overivrighet og stahet i startfasen er nok også en faktor i at utviklingen har gått som den har gått, hvor jeg har vært ubøyelig i min tro på at dette kunne fungere – som ren evolusjon – når jeg kanskje skulle fokusert på andre områder – som implementasjonen av det evolusjonære nettverket.

A Kildekode i Java

Kontroll.java

```
import java.io.*;

/**
 * Dette er klassen som starter det hele, hvor man kan velge mellom å kjøre evolusjonsalgoritmen
 * eller kjøre selv.
 *
 * @author Christian A. Myrvold
 * @version 09.08.08
 */
public class Kontroll
{
    Robot robot;

    /**
     * Klassens konstruktør. Kjører bare menyen.
     */
    public Kontroll()
    {
        menyValg();
    }

    /**
     * Skriv ut en meny, med valg, for de ulike tingene man skal kunne
     * gjøre med roboten.
     */
    public void menyValg()
    {
        velkommen();
        robot = new Robot();

        BufferedReader tast = new BufferedReader(new InputStreamReader(System.in));
        try
        {
            int valg = 100;
            do
            {
                System.out.print(meny());
                valg = new Integer(tast.readLine()).intValue();
                velgKontroll(valg);
                System.out.println();
            }
            while(valg != 0);
            String avslutt = "\n***Takk_for_at_du_valgte_CAMWorks'_robotkontroll.***\n";
            System.out.println(avslutt);
            robot.avslutt();
        }
        catch(Exception e)
        {
            e.printStackTrace();
            //System.out.println("Du må taste inn et tall i menyen");
            robot.avslutt();
        }
    }

    /**
     * En liten velkomsthilsen til brukeren.
     */
```

```
private void velkommen()
{
    String msg = "Velkommen til CAMWorks' robotkontroll. Vennligst gjør dine valg fra menyen.\n";
    System.out.println(msg);
}

/**
 * Menyen.
 */
private String meny()
{
    String meny = "";
    meny += "1) Kjør selv.\n";
    meny += "2) Kjør med evolusjon (så fort og rett frem som mulig/unngå objekter).\n\n";

    meny += "0) Avslutt.\n";
    meny += "> ";

    return meny;
}

/**
 * Gjør et valg ut fra brukerens ønske.
 */
private void velgKontroll(int valg)
{
    switch(valg)
    {
        case 0: break;
        case 1: new EksternKontroll(robot); break;
        case 2: velgSim(); break;
        default: System.out.println("Ikke gyldig valg. Prøv igjen."); break;
    }
}

/**
 * Skriv ut menyen.
 */
private void menySim()
{
    String meny = "\nVelg fra menyen:\n";
    meny += "1) Kjør online evolusjon.\n";
    meny += "2) Kjør offline (simulering).\n";
    meny += "0) Gå tilbake\n";
    meny += "> ";
    System.out.print(meny);
}

/**
 * Få input fra bruker i forhold til menySim.
 */
private void velgSim()
{
    BufferedReader tast = new BufferedReader(new InputStreamReader(System.in));
    try
    {
        int valg = 100;
        do
        {
            menySim();
            valg = new Integer(tast.readLine()).intValue();
            switch(valg)
```

```
                {
                    case 1: new EA(false, robot); break;
                    case 2: new EA(true, robot); break;
                    case 0: break;
                    default: System.out.println("Ikke gyldig valg. Prøv igjen."); break;
                }
            }
            while(valg != 0);

        }
        catch(Exception e)
        {
            e.printStackTrace();
        }
    }

}
```

EksternKontroll.java

```
import java.io.*;
import java.util.*;

/**
 * EksternKontroll er bare en liten klasse for styring av roboten via en meny, først og fremst
 * for testing. Her kan man også få printet ut forskjellige verdier som kan være nyttige.
 *
 * @author Christian A. Myrvold
 * @version 09.08.08
 */
public class EksternKontroll
{
    private Robot oscar;
    private int antallValg = 0;

    /**
     * Konstruktør for klassen. Lag et robotobjekt og skriv ut menyen.
     */
    public EksternKontroll(Robot r)
    {
        oscar = r;
        menyValg();
        oscar.avslutt();
    }

    /**
     * Standard menyvalg, hvor bruker må skrive et tall for å velge hva roboten skal gjøre.
     */
    public void menyValg()
    {
        BufferedReader tast = new BufferedReader(new InputStreamReader(System.in));
        try
        {
            int valg = 100;
            do
            {
                System.out.print(meny());
                valg = new Integer(tast.readLine()).intValue();
                int teller = 0;
                velgKontroll(valg, false, 0);
                System.out.flush();
            }
            while(valg != 0);
            String avslutt = "\nTakk for at du valgte CAMWorks' robotkontroll.\n";
            System.out.println(avslutt);
        }
        catch(Exception e)
        {
            System.out.println("Du må taste inn et tall i menyen");
        }
    }

    /**
     * Standard switch for kontroll, bestemmer hvilke metoder som skal kjøres i forhold
     * til brukerens valg.
     */
    private void velgKontroll(int valg, boolean tilf, int sek)
    {
        switch(valg)
        {
```

```
            case 1: if(! tilf ){ sek = beOmSekunder();} oscar.kjørRettFrem(sek); break;
            case 2: if(! tilf ){ sek = beOmSekunder();} oscar.kjørBakover(sek); break;
            case 3: oscar.svingSkarptVenstre90(); break;
            case 4: oscar.svingSkarptHøyre90(); break;
            case 5:  tilfeldig (); break;
            case 6: System.out.println(oscar); break;
            case 7: //oscar.backtrack(); break;
            case 0: break;
            default: System.out.println("'"+valg+"' er ikke et menyvalg. Prøv igjen"); break;
        }
    }

    /**
     * Be om antall sekunder fra bruker, typisk for å angi antall sekunder roboten skal kjøre.
     */
    private int beOmSekunder()
    {
        System.out.print("Vennligst tast inn antall sekunder: ");
        BufferedReader bf = new BufferedReader(new InputStreamReader(System.in));
        int sekunder = 0;
        try
        {
            sekunder = new Integer(bf.readLine()).intValue();
        }
        catch(Exception e)
        {
            System.out.println("Du må taste inn et heltall.");
        }
        return sekunder;
    }



    /**
     * Selve menyen som en string.
     */
    private String meny()
    {
        int teller = 0;
        String meny = "";
        meny += ""+ ++teller + ") Kjør fremover i x antall sekunder.\n"; //1
        meny += ""+ ++teller + ") Kjør bakover i x antall sekunder.\n"; //2
        meny += ""+ ++teller + ") Sving til venstre.\n";                 //3
        meny += ""+ ++teller + ") Sving til høyre.\n";                   //4
        meny += "5) Kjør tilfeldig.\n";
        meny += "6) Print ut verdier fra sensorer.\n";
        meny += "7) Roter tilbake.\n";
        meny += "0) Avslutt\n";
        meny += "> ";
        antallValg = ++teller;
        return meny;
    }

    /**
     * Tilfeldig styring av roboten. Brukes for testing, slik at man kan se mange
     * forskjellige handlinger på kort tid.
     */
    private void tilfeldig ()
    {
        Random rand = new Random();
        Random rSek = new Random();
        int tilf = 100;
```

```
        do
        {
            tilf  = rand.nextInt(antallValg);
            velgKontroll( tilf ,  true, rSek.nextInt (3));
        }
        while(tilf != 0);
    }

}
```

Robot.java

```
//importer iCommand slik at jeg kan kommunisere med roboten via Bluetooth.
//import lejos.nxt.*;
import icommand.nxt.comm.NXTCommand;
import icommand.nxt.*;
import icommand.navigation.*;

import java.lang.Math;

/**
 * Denne klassen inneholder alle metoder og felt som har med styring av roboten og dens
 * sensorer å gjøre. Inkludert er også flere felt for sensorer som jeg ikke lenger tar i
 * bruk, slik at man kan se hvordan det gjøres.
 *
 * @author Christian Andreas Myrvold
 * @version 09.08.08
 */
public class Robot
{
    private Motor venstre;
    private Motor høyre;
    private Pilot pilot;
    private TouchSensor trykk;
    private Battery batteri;
    private UltrasonicSensor hode;
    private LightSensor lys;
    private SoundSensor lyd;
    private CompassSensor compass;
    private int fart = 720;
    private int maxFart = 900;
    private Kart kart;

    /**
     * Konstruktør for klassen Robot. Den initialiserer alle sensorene og setter
     * initialfarten til motorene.
     */
    public Robot()
    {
        kobleTil();
        venstre = Motor.A;
        høyre = Motor.C;
        hode = new UltrasonicSensor(SensorPort.S4);
        lys = new LightSensor(SensorPort.S3);
        hode.setMetric(true); //sett til metermåling
        venstre.setSpeed(maxFart);
        høyre.setSpeed(maxFart);

        //pilot = new Pilot(2.1f, 5.5f, venstre, høyre);
        //trykk = new TouchSensor(SensorPort.S1);
        //batteri = new Battery();
        //compass = new CompassSensor(SensorPort.S2);
        //compass.startCalibration(); //kalibrer kompasser for mer nøyaktig måling
        //lyd = new SoundSensor(SensorPort.S2);
        //pilot.setSpeed(fart);
        //venstre.smoothAcceleration(true);
        //høyre.smoothAcceleration(true);
        //test();
    }

    /**
     * Koble til roboten med bluetooth, og gi en OK hvis dette går bra.
```

```
 */
private void kobleTil()
{
    NXTCommand.open();
    NXTCommand.setVerify(true);
    String msg = "Tilkoblet robot. Klar til å kjøre :)";
    System.out.println(msg);
}

/**
 * Kjør rett fremover i antall sekunder. Tror dette er den beste måten å gjøre det på, så kan jeg
 * bruke en løkke for å regulere hvor lenge den får lov til å kjøre.
 */
public void kjørRettFrem(int sekunder)
{
    //pilot.travel(100);
    try
    {
        //pilot.forward();
        sekunder = 450;
        venstre.forward();
        høyre.forward();

        Thread.sleep(sekunder);
        //pilot.stop();
        høyre.stop();
        venstre.stop();
    }
    catch(Exception e)
    {
        System.out.println("Kunne ikke kjøre fremover.");
    }
}

/**
 * Kjør rett bakover i antall sekunder.
 */
public void kjørBakover(int sekunder)
{
    try
    {
        //pilot.backward();
        venstre.backward();
        høyre.backward();
        Thread.sleep(sekunder*1000);
        //pilot.stop();
        høyre.stop();
        venstre.stop();
    }
    catch(Exception e)
    {
        System.out.println("Kunne ikke kjøre bakover.");
    }
}

/**
 * Sving skarpt til venstre 90 grader. Dette innebærer at høyre motor
 * kjører bakover mens venstre kjører fremover i 275 millisekunder.
 * Vanskelig å få til ordentlig.
 */
public void svingSkarptVenstre90()
{
```

```
    try
    {
        /*
        høyre.rotateTo(45);
        høyre.forward();
        Thread.sleep(800);
        høyre.stop();*/
        høyre.forward();
        venstre.backward();
        Thread.sleep(275);
        høyre.stop();
        venstre.stop();
    }
    catch(Exception e)
    {
    }
}

/**
 * Sving skarpt til høyre 90 grader. Ellers samme som metoden ovenfor.
 */
public void svingSkarptHøyre90()
{
    try
    {
        /*
        //venstre.rotateTo(45);
        venstre.forward();
        Thread.sleep(800);
        venstre.stop();*/
        venstre.forward();
        høyre.backward();
        Thread.sleep(250);
        høyre.stop();
        venstre.stop();
    }
    catch(Exception e)
    {}
}


/**
 * SENSORER.
 */

/**
 * Batterimåler. Ikke akkurat en sensor, men jeg tenkte å bruke den slik at batteristyrken kan påvirke
 * kontrolleringen av roboten. Maxverdi = 9V
 */
public double batteriMåler()
{
    int max = 9;
    return Battery.getVoltage()/max;
}

/**
 * TRYKKSENSOR. Den store trykktesten. Sjekk trykksensoren for input mens roboten kjører. Hvis den trykkes inn,
 * velg handling. Denne handlingen kan endres dynamisk.
 * Merk: Dette ble bare brukt i test, senere ble trykksensor fjernet.
 */
private void sjekkTrykk()
{
```

```
    while(høyre.isMoving() || venstre.isMoving())
    {
        if(trykk.isPressed())
        {
            kjørBakover(2);
        }
    }
}

/**
 * ULTRASONIC SENSOR. Med denne kan jeg beregne distanse til nærmeste
 * objekt, i centimeter. Max distanse er 255 cm, +/- 3 cm. Returnerer
 * normalisert verdi.
 */
public double distanse()
{
    double max = 255.0;
    double distanse = (double)hode.getDistance();
    return distanse/max;
}

/**
 * Samme som distanse(), men hvor verdien kommer ut som centimeter.
 */
public int distanseCM()
{
    return hode.getDistance();
}

/**
 * LYSSENSOR. Få lysstyrke både i verdi og prosent. Begge kan være nyttig,
 * så jeg lager en hjelpemetode for begge disse. Kan bli særlig nyttig når
 * jeg skal normalisere verdiene.
 */
public int lysstyrkeVerdi()
{
    lys.passivate();
    return lys.getLightValue();
}

/**
 * Lysstyrke i prosent.
 */
public int lysstyrkeProsent()
{
    lys.passivate();
    return lys.getLightPercent();
}

/**
 * LYDSENSOR. Returnerer lyden som mikrofonen får inn i desibel (dB).
 */
public int lydStyrke()
{
    return lyd.getdB();
}

/**
 * COMPASS SENSOR. Returns the degrees it has gone from the beginning.
 */
public double degrees()
{
```

```
    compass.startCalibration(); //kalibrer kompasset for mer nøyaktig måling
    //return compass.getDegrees();
    return compass.getDegreesCartesian();
}

/**
 * Få ut en enklere kompassavlesning for bruk i kartet. Denne metoden skal
 * returnere et tall mellom 1-8, i forhold til 8 himmelretninger (N, NØ, Ø,
 * SØ, S, SV, V, NV).
 */
public int himmelretning()
{
    final int antallRetninger = 8;
    int grader = (int)degrees();
    while(grader > 360)
    {
        grader -= 360;
    }

    int retning = 1;
    int grense = 360/antallRetninger;
    while(grader > grense)
    {
        retning++;
        grader -= grense;
    }
    return retning;
}

/**
 * Utfør i forhold til betydning og aktiveringsstyrke for begge motorene.
 * Regner også ut fitnessen for fremdriften.
 */
public int kjør(double aktiv, double aktiv2)
{
    double fremover = 0.55;
    double stop = 0.05;
    double halfway = 0.225;
    int vent = 300; //i millisekunder
    int fitness = 0;
    //motor.flt();
    Motor en = venstre;
    Motor to = høyre;


    if(aktiv > fremover) // aktiv > 0.55
    {
        en.setSpeed(regnUt(aktiv));
        en.forward();

        if(aktiv > halfway+fremover) //aktiv > 0.775
        {
            fitness++;
        }
    }
    else if(aktiv < (fremover-stop) && aktiv > stop) //0.05 < aktiv < 0.5
    {
        en.setSpeed(regnUt(aktiv+0.5));
        en.backward();

        if(aktiv > halfway+stop) //aktiv > 0.275
        {
```

```
            fitness++;
        }
    }

    if(aktiv2 > fremover) //aktiv2 < 0.55
    {
        to.setSpeed(regnUt(aktiv2));
        to.forward();

        if(aktiv2 > halfway+fremover) //aktiv2 > 0.775
        {
            fitness++;
        }
    }
    else if(aktiv2 < (fremover-stop) && aktiv > stop) //0.05 < aktiv2 < 0.5
    {
        to.setSpeed(regnUt(aktiv2+0.5));
        to.backward();

        if(aktiv2 > halfway+stop) //aktiv2 > 0.275
        {
            fitness++;
        }
    }

    //test sensorene for hindre foran og bak
    int grense_distanse = 13;
    double grense_lys = 0.25;

    if(distanse() > 0.5)
    {
        fitness++;
    }
    if(lysstyrkeProsent() > 0.5)
    {
        fitness++;
    }

    try
    {
        Thread.sleep(vent);
        en.stop();
        to.stop();
        //en.flt();
        //to.flt();
    }
    catch(Exception e)
    {
        e.printStackTrace();
    }
    return fitness;
}

/**
 * Regn ut farten på motoren i forhold til styrken.
 */
private int regnUt(double styrke)
{
    int svar = (int)Math.round(styrke * maxFart);
    return svar;
}
```

```
/**
 * En egen metode for å avslutte, som egentlig bare betyr at jeg må lukke
 * kommunikasjonen med roboten.
 */
public void avslutt()
{
    NXTCommand.close();
}

/**
 * toString-metoden, her tenker jeg å printe ut diverse statistikk og
 * sensormålinger.
 */
public String toString()
{
    String melding = "Batteri (V):\t"+batteriMåler()+"\n";
    melding += "UltrasonicSensor (%):\t"+distanse()+"\n";
    melding += "UltrasonicSensor (cm):\t"+distanseCM()+"\n";
    lys.passivate();
    //lys.activate();
    melding += "LightSensor (%):\t"+lysstyrkeProsent()+"\n";
    melding += "LightSensor (value):\t"+lysstyrkeVerdi()+"\n";
    //lys.passivate();
    //melding += "SoundSensor (dB): \t"+lydStyrke()+"\n";
    melding += "Motor Tacho (V):\t"+venstre.getTachoCount()+"\n";
    melding += "Motor Tacho (H):\t"+høyre.getTachoCount()+"\n";
    melding += "Compass degrees:\t"+degrees()+"\n";
    reset();
    return melding;
}

/**
 * Reset forskjellige blokker.
 */
public void reset()
{
    høyre.resetTachoCount();
    venstre.resetTachoCount();
    compass.resetCartesianZero();
}

/**
 * Get the distance traveled by the left motor.
 */
public int getLeftMotorTacho()
{
    return venstre.getBlockTacho();
}

/**
 * Get the distance traveled by the right motor.
 */
public int getRightMotorTacho()
{
    return høyre.getBlockTacho();
}

/**
 * Hent kartet fra roboten. Dette er først og fremst for at fitnessfunksjonen
 * skal kunne få tak i det.
 */
public Kart hentKart()
```

```
{
    return kart;
}

/**
 * Oppdater det interne kartet i roboten.
 */
public void oppdaterKart(Kart k)
{
    kart = k;
}

/**
 * Backtrackingmetode. Denne metoden skulle bli brukt for å få roboten tilbake til start-
 * punktet i begynnelsen av generasjonen. Hvert kromosom må evalueres fra det samme rom-
 * punktet som resten, ellers vil validiteten til testen ødelegges. Handlingslisten må
 * reverseres (til en viss grad) siden roboten må kopiere de siste handlingene som ble
 * utført først.
 * */
public void backtrack(double[] actions)
{
    int length = actions.length;
    for(int i = 0; i < length; i += 2)
    {
        double left_act = actions[length-(i+2)];
        double right_act = actions[length-(i+1)];
        kjørTacho(left_act, right_act);
    }
    //reset();
}


/**
 * Kjør så mange rotasjoner som blokktachoen angir.
 * Fungerer ikke helt på grunn av motorene.
 */
private void kjørTacho(double left_act, double right_act)
{
    double fremover = 0.50;
    double stop = 0.05;
    int vent = 300; //i millisekunder
    //motor.flt();
    double aktiv = left_act;
    double aktiv2 = right_act;


    if(left_act < fremover)
    {
        aktiv += fremover;
    }
    else
    {
        aktiv -= fremover;
    }

    if(right_act < fremover)
    {
        aktiv2 += fremover;
    }
    else
    {
        aktiv2 -= fremover;
```

```
            }
            kjør(aktiv, aktiv2);
        }

        /**
         * Et annet forsøk på å få kjørTacho til å fungere, men det
         * vil seg ikke.
         */
        private void kjørTacho(int left_act, int right_act)
        {
            høyre.rotateTo(right_act, true);
            venstre.rotateTo(left_act, true);
            reset();
        }
}
```

RobotFitness.java

```
import org.jgap.*;
import org.jgap.impl.*;

import java.lang.Math;
import java.io.*;

/**
 * Fitnessfunksjonen til roboten. Denne må legges i en egen subklasse som utvides
 * fra FitnessFunction-klassen i JGAP.
 *
 * @author Christian A. Myrvold
 * @version 09.08.08
 */
public class RobotFitness extends FitnessFunction
{
    private Robot robot;
    private int generation;
    private Kart kart;

    /**
     * Konstruktøren for klassen, initialiserer bare noen felt.
     */
    public RobotFitness(Robot r)
    {
        robot = r;
        generation = 0;
    }

    /**
     * Evalueringsmetoden for å kjøre i en sirkulær hinderbane og unngå vegger og hindre.
     * All evolusjon skal skje på kartet, og bare det beste individet skal utføres direkte
     * på roboten.
     */
    public double evaluate(IChromosome chromosome)
    {
        int fitness = 0;
        int maxOmdreininger = 300;
        Gene[] genes = chromosome.getGenes();
        kart = robot.hentKart();

        int retning = robot.himmelretning();
        for(int i = 0; i < genes.length; i += 2)
        {
            double venstre_str = ((DoubleGene) genes[i]).doubleValue();
            double høyre_str = ((DoubleGene) genes[i+1]).doubleValue();
            int venstre_fart = (int)venstre_str*maxOmdreininger;
            int høyre_fart = (int)høyre_str*maxOmdreininger;

            int nyRetning = nyRetning(retning, venstre_fart, høyre_fart);
            int hjulFart = finnFart(venstre_fart, høyre_fart);
            fitness += kart.simuler(nyRetning, hjulFart);
            fitness += beregnFartFitness(venstre_str, høyre_str);
        }
        fitness /= genes.length/2;
        return fitness;
    }

    /**
     * Beregn fitness for farten i simuleringen på kartet. Dette er bare en enkel
     * sjekk på om verdiene går over en viss grense. Ønsker å gi mer fitness for
```

```
 * fremoverdrift.
 */
private int beregnFartFitness(double hjul_en, double hjul_to)
{
    double grense_bakover = 0.275;
    double grense_fremover = 0.775;
    double halvveis = 0.5;
    int fitness = 0;

    //sjekk første hjul
    if(hjul_en > grense_bakover && hjul_en < halvveis)
    {
        fitness++;
    }
    else if(hjul_en > grense_fremover)
    {
        fitness++;
    }

    //sjekk andre hjul
    if(hjul_to > grense_bakover && hjul_to < halvveis)
    {
        fitness++;
    }
    else if(hjul_to > grense_fremover)
    {
        fitness++;
    }
    return fitness;
}



/**
 * Finn den forenklete farten i forhold til input.
 * fart = 0; roboten går ikke
 * fart = 1; roboten går fremover
 * fart = 2; roboten går bakover
 */
private int finnFart(int venstre, int høyre)
{
    int fart = 0;
    int fremover = 1;
    int bakover = 2;
    int svingSvakt = 100;
    int svingSvaktBak = -100;
    if(venstre > svingSvakt && høyre > svingSvakt)
    {
        fart = fremover;
    }
    else if(venstre < svingSvaktBak && høyre > svingSvaktBak)
    {
        fart = bakover;
    }
    return fart;
}


/**
 * Kalkuler retningen i forhold til hjulfarten på begge hjulene. Dette blir ikke helt
 * nøyaktig, men burde være nære nok for simuleringens skyld.
 *
```

```
 * Ikke bare bare, siden forskjellige variasjoner av fart på hjulene kan resultere i samme
 * retning, så det blir mye å forholde seg til.
 * Metoden blir veldig tilfeldig på grunn av motorene.
 */
private int nyRetning(int retning, double venstre_fart, double høyre_fart)
{
    int svingHardt = 200;
    int svingSvakt = 100;
    int svingVeldigSvakt = 50;
    int svingHardtBak = -200;
    int svingSvaktBak = -100;
    int svingVeldigSvaktBak = -50;

    if (!(( venstre_fart > svingSvakt && høyre_fart > svingSvakt) ||
    (venstre_fart < svingSvaktBak && høyre_fart < svingSvaktBak))) //kjører fremover eller bakover
    {
        if (((venstre_fart > svingSvakt && høyre_fart < svingSvaktBak)) ||
        (venstre_fart > svingHardt && høyre_fart < svingSvakt) ||
        (høyre_fart < svingHardtBak && venstre_fart < svingSvakt)) //svinger hardt til høyre
        {
            retning = leggTilRetning(retning);
            retning = leggTilRetning(retning);
        }

        else if((høyre_fart > svingSvakt && venstre_fart < svingSvaktBak) ||
        (høyre_fart > svingHardt && venstre_fart < svingSvakt) ||
        (venstre_fart < svingHardtBak && høyre_fart < svingSvakt))//svinger hardt til venstre
        {
            retning = trekkFraRetning(retning);
            retning = trekkFraRetning(retning);
        }
        else if((venstre_fart > svingSvakt && høyre_fart > svingSvaktBak) ||
        (venstre_fart > svingSvakt && høyre_fart < svingVeldigSvakt) ||
        (høyre_fart < svingSvaktBak && venstre_fart > svingVeldigSvaktBak)) //sving svakt til høyre
        {
            retning = leggTilRetning(retning);
        }
        else if((høyre_fart > svingSvakt && venstre_fart > svingSvaktBak) ||
        (høyre_fart > svingSvakt && venstre_fart < svingVeldigSvakt) ||
        (venstre_fart < svingSvaktBak && høyre_fart > svingVeldigSvaktBak)) //sving svakt til venstre
        {
            retning = trekkFraRetning(retning);
        }
    }
    return retning;
}

/**
 * Hjelpemetode for at retningen ikke skal gå utenfor grensene. 1 <= retning <=8.
 * Trekk fra én i retning.
 */
private int trekkFraRetning(int retning)
{
    retning--;
    if(retning < 1)
    {
        retning = 8;
    }
    return retning;
}

/**
```

```
     * Legg til én i retning.
     */
    private int leggTilRetning(int retning)
    {
        retning++;
        if(retning > 8)
        {
            retning = 1;
        }
        return retning;
    }

    /**
     * Hent ut kartet. Denne må brukes siden fitnessfunksjonen bruker kartet, men som skal kunne endres andre
     * steder.
     */
    public Kart hentKart()
    {
        return kart;
    }

    /**
     * Endre på kartet. Denne må brukes siden kartet også brukes og endres utenom denne klassen,
     */
    public void endreKart(Kart k)
    {
        kart = k;
    }
}
```

EA.java

```
import org.jgap.*;
import org.jgap.impl.*;

import java.util.*;
import java.lang.Math;
import java.io.*;

/**
 * Dette er klassen hvor evolusjonen vil bli gjort. Den
 * bruker JGAP og tolker resultatene.
 *
 * @author Christian A. Myrvold
 * @version 09.08.08
 */
public class EA
{
    private Robot robot;
    private Population[][] solutions;
    private int [][] antallBrukt;
    private Kart kart = new Kart();


    /**
     * Konstruktøren for klassen, starter en ny populasjon
     * og initialiserer forskjellige felt.
     */
    public EA(boolean sim, Robot r)
    {
        robot = r;
        //solutions = new Population[100][100];
        solutions = new Population[10][10];
        antallBrukt = new int[10][10];
        kjørOgUnngå();
    }


    /**
     * Metoden for å unngå hindre mens den kjører.
     * Simuler forskjellige løsninger og utfør det
     * beste individet.
     */
    public void kjørOgUnngå()
    {
        int nrIndividuals = 10;
        int chromosomeSize = 2;
        int mutationRate = 50;
        double crossoverRate = 0.8;

        //Endre konfigurasjonen for dette problemet.
        Configuration.reset();
        Configuration config = new DefaultConfiguration();
        config.setPreservFittestIndividual(true);
        config.setKeepPopulationSizeConstant(true);

        Genotype genotype = null;
        try
        {
            IChromosome sample = new Chromosome(config, new DoubleGene(config, 0.0, 1.0), chromosomeSize);
            config.setSampleChromosome(sample);
            config.setPopulationSize(nrIndividuals);
```

```
        config.setFitnessFunction(new RobotFitness(robot));
        config.addGeneticOperator(new CrossoverOperator(config, crossoverRate));
        config.addGeneticOperator(new MutationOperator(config, mutationRate));
        config.addNaturalSelector(new BestChromosomesSelector(config), false);
        //kan ikke bruke WeightedRouletteSelector, får bare NullPointerException fra JGAP
        //config.addNaturalSelector(new WeightedRouletteSelector(config), false);
        config.setRandomGenerator(new GaussianRandomGenerator());
        genotype = Genotype.randomInitialGenotype(config);
    }
    catch(InvalidConfigurationException e)
    {
        e.printStackTrace();
    }

    //sett i gang evolusjonen
    int generasjoner = 0;
    int maxGenerasjoner = 10;
    int evolveGenerasjoner = 5;
    int retning = 1;

    //sett hindrene roboten kan "se" før den har begynt å bevege seg
    //kart.setObstacle(robot.himmelretning(), robot.distanseCM(), robot.lysstyrkeProsent());
    kart.setObstacle(retning, robot.distanseCM(), robot.lysstyrkeProsent());
    robot.oppdaterKart(kart);
    kart.printKart();

    //play(config);
    while(generasjoner < maxGenerasjoner)
    {
        System.out.println("Generasjon:␣"+ (generasjoner+1));

        //find out if there's a solution already for these sensor-readings
        double us = robot.distanse();
        int ls = robot.lysstyrkeProsent();
        String usX = "" + us;
        //String lsY = "" + ls;
        String usTest = "";
        if(!usX.equals("1.0"))
        {
            usTest = usX.substring(0, 3);
            //usTest = usX.substring(0, 4);
        }
        else
        {
            usTest = usX;
        }
        //int x = (int)((new Double(usTest)).doubleValue()*100)-1;
        //int y = ls-1;
        int x = (int)((new Double(usTest)).doubleValue()*10);
        if(x == 10)
        {
            x = 9;
        }
        double mellom = ls/10;
        int y = ((int)Math.floor(mellom));

        System.out.println("****TESTVERDIER****");
        System.out.println(robot);

        //splitt opp, én for når det er en løsning, én for når det ikke finnes en løsning
        FitnessFunction myFF = config.getFitnessFunction();
        Population defender = solutions[x][y];
```

```
if(defender != null)
{
    antallBrukt[x][y]++;
    IChromosome defenderFittest = defender.determineFittestChromosome();
    double bestFitnessBefore = defenderFittest.getFitnessValue();
    //utfør dette individet og test fitness etterpå
    //hvis fitness ikke er optimal lenger, evolver denne populasjonen X generasjoner
    double bestFitnessNow = executeChromosome(defenderFittest);

    if(bestFitnessNow < bestFitnessBefore)
    {
        try
        {
            genotype = null;
            genotype = new Genotype(config, defender);
            genotype.evolve(evolveGenerasjoner);
            defender = genotype.getPopulation();
            //evolver denne populasjonen
        }
        catch(InvalidConfigurationException e)
        {
            e.printStackTrace();
        }
    }
    else
    {
        //oppdater kromosomet
        for(int i = 0; i < defender.size(); i++)
        {
            if(defender.getChromosome(i).equals(defenderFittest))
            {
                defenderFittest.setFitnessValue(bestFitnessNow);
                defender.setChromosome(i, defenderFittest);
            }
        }

    }

    //lagre den evolverte populasjonen i dobbelarrayet
    solutions[x][y] = defender;
}
else
{
    antallBrukt[x][y]++;
    try
    {
        //make a new population
        genotype = null;
        genotype = Genotype.randomInitialGenotype(config);
    }
    catch(InvalidConfigurationException e)
    {
        e.printStackTrace();
    }

    //evolve this genotype
    genotype.evolve(evolveGenerasjoner);

    Population pop = genotype.getPopulation();
    IChromosome fittest = pop.determineFittestChromosome();
    pop.sortByFitness();
    List<IChromosome> liste = pop.getChromosomes();
```

```
            //FOR DEBUGGING
            double fitness = 0.0;
            double sumFitness = 0.0;
            double bestFitness = 0.0;
            double worstFitness = 0.0;
            IChromosome bestChromosome = null;
            IChromosome worstChromosome = null;
            int counter = 0;
            for(IChromosome kromosom : liste)
            {
                fitness  = kromosom.getFitnessValue();
                sumFitness += fitness;

                if(counter == 0)
                {
                    bestFitness = fitness;
                    bestChromosome = kromosom;
                }
                else if(counter == nrIndividuals-1)
                {
                    worstFitness = fitness;
                    worstChromosome = kromosom;
                }
                counter++;
            }
            double gjFitness = sumFitness/nrIndividuals;
            System.out.println("Best fitness: "+bestFitness);
            System.out.println("Oppbygning: "+bestChromosome);
            System.out.println("Worst fitness: "+worstFitness);
            System.out.println("Oppbygning: "+worstChromosome);
            System.out.println("Gjennomsnittlig fitness: "+gjFitness+"\n");



            //utfør beste individ
            solutions[x][y] = pop;
            executeChromosome(fittest);
        }
        generasjoner++;

        //oppdater kartet i FitnessFunksjonsklassen også
        robot.oppdaterKart(kart);

        //print ut kartet for å se om det samsvarer med "virkeligheten"
        kart.printKart();
    }
    //print ut antall ganger hver populasjoner er brukt.
    //for debugging.
    for(int i = 0; i < antallBrukt.length; i++)
    {
        for(int j = 0; j < antallBrukt.length; j++)
        {
            int nr = (i*10)+j;
            System.out.println("Populasjon nr: "+nr+" brukt "+antallBrukt[i][j]+" ganger.");
        }
    }
}

/**
 * Tanken om å "leke", som egentlig bare betydde å kjøre rundt tilfeldig . Løsningene skulle
 * ikke evolveres, men bare huske hva som skjer. Dette betyr at hver handling hadde blitt
```

```
     * testet og lagret (hvis det er den beste løsningen så langt) i løsningsarrayen. Populasjonen
     * vil være separat fra de evolvert, og vil bare utføres.
     */
    private void play(Configuration config)
    {
        //create the new genotype
        Genotype genotype = null;
        try
        {
            genotype = Genotype.randomInitialGenotype(config);
        }
        catch(InvalidConfigurationException e)
        {
            e.printStackTrace();
        }

        FitnessFunction myFF = config.getFitnessFunction();
        Population pop = genotype.getPopulation();

        Iterator iter = null;
        for(iter = pop.iterator(); iter.hasNext(); )
        {
            IChromosome ic = (IChromosome)iter.next();
            executeChromosome(ic);
        }
    }

    /**
     * Utfør handlingene til kromosomet.
     */
    private double executeChromosome(IChromosome chromo)
    {
        Gene[] genes = chromo.getGenes();
        double fitness = 0;
        for(int i = 0; i < genes.length; i += 2)
        {
            double left_act = ((DoubleGene) genes[i]).doubleValue();
            double right_act = ((DoubleGene) genes[i+1]).doubleValue();
            fitness += robot.kjør(left_act, right_act);

            //oppdater kartet
            kart.setObstacle(robot.himmelretning(), robot.distanseCM(), robot.lysstyrkeProsent());
        }
        return fitness;
    }
}
```

Kart.java

```
/**
 * Et 2D-kart som roboten skal kunne navigere etter. Den skal registrere
 * hvor roboten er, og etter hvert registrere hindre, som f.eks. vegg og
 * naturlige hindre i veien. Prøver først med en int-dobbelarray.
 *
 * @author Christian A. Myrvold
 * @version 09.08.08
 */
public class Kart
{
    private boolean[][] kart;
    private int vidde = 4;
    private int bredde = 4;
    private int nrRehash = 1;
    private int retning = 1; //fra 1-8 i forhold til himmelretning
    //x- og y-koordinatene til roboten
    int x = 2;
    int y = 2;

    //Himmelretninger som tall
    final int NORD = 1;
    final int NORD_ØST = 2;
    final int ØST = 3;
    final int SØR_ØST = 4;
    final int SØR = 5;
    final int SØR_VEST = 6;
    final int VEST = 7;
    final int NORD_VEST = 8;

    /**
     * Konstruktør for klassen Kart, initialiserer bare kartet.
     */
    public Kart()
    {
        kart = new boolean[vidde][bredde];
    }

    /**
     * Print ut kartet, slik at man kan lettere kan se om den er bra nok.
     * Husk også å vise roboten på kartet.
     */
    public void printKart()
    {
        System.out.println("****KART****");
        System.out.println();
        System.out.println("-----------------------------------------");
        for(int i = 0; i < vidde; i++)
        {
            for(int j = 0; j < bredde; j++)
            {
                if(kart[i][j])
                {
                    System.out.print("| X ");
                }
                else if(x == i && y == j)
                {
                    System.out.print("| R ");
                }
                else
```

```
            {
                System.out.print("|   ");
            }

        }
        System.out.println("|\n-----------------------------------------------------------");
    }
}

/**
 * Lag et nytt kart, doble størrelsen, og få de gamle verdiene inn i
 * det nye kartet. Oppdater x- og y-koordinatene, som er egne globale
 * variable.
 */
private void rehash()
{
    nrRehash++;
    int oldVidde = vidde;
    int oldBredde = bredde;
    vidde *= 2;
    bredde *= 2;
    boolean[][] oldKart = kart;
    kart = new boolean[vidde][bredde];

    for(int i = 0; i < oldVidde; i++)
    {
        for(int j = 0; j < oldBredde; j++)
        {
            if(oldKart[i][j])
            {
                kart[i+nrRehash][j+nrRehash] = true;
            }
        }
    }

    //oppdater koordinatene til roboten
    x += nrRehash;
    y += nrRehash;
}

/**
 * Beregn distansen i kartmål, slik at det er snakk om antall ruter
 * istedenfor i centimeter.
 */
private int hentKartDistanse(int distanse)
{
    int grense = 13;
    int kartDistanse = 5;
    if(distanse < grense)
    {
        kartDistanse = 1;
    }
    else if(distanse < grense*2)
    {
        kartDistanse = 2;
    }
    else if(distanse < grense*3)
    {
        kartDistanse = 3;
    }
    else if(distanse < grense*4)
    {
```

```
        kartDistanse = 4;
    }
    return kartDistanse;
}

/**
 * Beregn distanse til hindre bak roboten gjennom lysstyrken.
 */
private int hentKartDistanseBakover(double lys)
{
    double grenseBakover_en = 20;
    double grenseBakover_to = 25;
    int distanse = 3;

    if(lys < grenseBakover_en)
    {
        distanse = 1;
    }
    else if(lys < grenseBakover_to)
    {
        distanse = 2;
    }
    return distanse;
}

/**
 * Sett en del av kartet til å være et hinder. Sjekk først retningen
 * til roboten, så distansen til hinderet, og kalkuler så hvilken kart-
 * del som skal settes som 'true'. Før hvert hinder settes, må det
 * sjekkes om det er innenfor kartets grenser, ellers må det rehashes.
 */
public void setObstacle(int nyRetning, int distanse, int lys)
{
    retning = nyRetning;
    int mDistanse = hentKartDistanse(distanse);
    int lDistanse = hentKartDistanseBakover(lys);

    if(retning == NORD)
    {
        //sjekk grensene
        while(y-mDistanse < 0)
        {
            rehash();
        }

        if(mDistanse == 1)
        {
            //sett hinderet
            kart[x][y-1] = true;
        }
        else
        {
            int count = 1;
            while(count < mDistanse)
            {
                //fjern hinderet
                kart[x][y-count] = false;
                count++;
            }
            //sett hinderet
            if(mDistanse != 5)
            {
```

```
                kart[x][y-count] = true;
            }
        }

        //sett hindre bak
        while(y+lDistanse > vidde-1)
        {
            rehash();
        }

        if(lDistanse == 1)
        {
            //sett hinderet
            kart[x][y+1] = true;
        }
        else if(lDistanse == 2)
        {
            //slett hinder
            kart[x][y+1] = false;
            //sett hinder
            kart[x][y+2] = true;
        }
        else //lDistanse == 0 (ingen hindre)
        {
            //slett hindre
            kart[x][y+1] = false;
            kart[x][y+2] = false;
        }

    }
    else if(retning == NORD_ØST)//NORTH-EAST
    {
        //sjekk grensene
        while(y-mDistanse < 0 || x+mDistanse > vidde-1)
        {
            rehash();
        }

        if(mDistanse == 1)
        {
            //sett hinderet
            kart[x+1][y-1] = true;
        }
        else
        {
            int count = 1;
            while(count < mDistanse)
            {
                //fjern hinderet
                kart[x+count][y-count] = false;
                count++;
            }
            //sett hinderet
            if(mDistanse != 5)
            {
                kart[x+count][y-count] = true;
            }
        }

        //sett hindre bak
        while(y+lDistanse > vidde-1 || x-lDistanse < 0)
        {
```

```
            rehash();
        }

        if(lDistanse == 1)
        {
            //sett hinderet
            kart[x-1][y+1] = true;
        }
        else if(lDistanse == 2)
        {
            //slett hinder
            kart[x-1][y+1] = false;
            //sett hinder
            kart[x-2][y+2] = true;
        }
        else
        {
            //slett hindre
            kart[x-1][y+1] = false;
            kart[x-2][y+2] = false;
        }
    }
    else if(retning == ØST)//EAST
    {
        //sjekk grensene
        while(x+mDistanse > vidde-1)
        {
            rehash();
        }

        if(mDistanse == 1)
        {
            //sett hinderet
            kart[x+1][y] = true;
        }
        else
        {
            int count = 1;
            while(count < mDistanse)
            {
                //fjern hinderet
                kart[x+count][y] = false;
                count++;
            }
            //sett hinderet
            if(mDistanse != 5)
            {
                kart[x+count][y] = true;
            }
        }

        //sett hindre bak
        while(x-lDistanse < 0)
        {
            rehash();
        }

        if(lDistanse == 1)
        {
            //sett hinderet
            kart[x-1][y] = true;
        }
```

```
    else if(lDistanse == 2)
    {
        //slett hinder
        kart[x-1][y] = false;
        //sett hinder
        kart[x-2][y] = true;
    }
    else
    {
        //slett hindre
        kart[x-1][y] = false;
        kart[x-2][y] = false;
    }
}
else if(retning == SØR_ØST)//SOUTH-EAST
{
    //sjekk grensene
    while(y+mDistanse > bredde-1 || x+mDistanse > vidde-1)
    {
        rehash();
    }

    if(mDistanse == 1)
    {
        //sett hinderet
        kart[x+1][y+1] = true;
    }
    else
    {
        int count = 1;
        while(count < mDistanse)
        {
            //fjern hinderet
            kart[x+count][y+count] = false;
            count++;
        }
        //sett hinderet
        if(mDistanse != 5)
        {
            kart[x+count][y+count] = true;
        }
    }

    //sett hindre bak
    while(y-lDistanse < 0 || x-lDistanse < 0)
    {
        rehash();
    }

    if(lDistanse == 1)
    {
        //sett hinderet
        kart[x-1][y-1] = true;
    }
    else if(lDistanse == 2)
    {
        //slett hinder
        kart[x-1][y-1] = false;
        //sett hinder
        kart[x-2][y-2] = true;
    }
    else
```

```
    {
        //slett hindre
        kart[x-1][y-1] = false;
        kart[x-2][y-2] = false;
    }
}
else if(retning == SØR)//SOUTH
{
    //sjekk grensene
    while(y+mDistanse > bredde-1)
    {
        rehash();
    }

    if(mDistanse == 1)
    {
        //sett hinderet
        kart[x][y+1] = true;
    }
    else
    {
        int count = 1;
        while(count < mDistanse)
        {
            //fjern hinderet
            kart[x][y+count] = false;
            count++;
        }
        //sett hinderet
        if(mDistanse != 5)
        {
            kart[x][y+count] = true;
        }
    }

    //sett hindre bak
    while(y-lDistanse < 0)
    {
        rehash();
    }

    if(lDistanse == 1)
    {
        //sett hinderet
        kart[x][y-1] = true;
    }
    else if(lDistanse == 2)
    {
        //slett hinder
        kart[x][y-1] = false;
        //sett hinder
        kart[x][y-2] = true;
    }
    else
    {
        //slett hindre
        kart[x][y-1] = false;
        kart[x][y-2] = false;
    }
}
else if(retning == SØR_VEST)//SOUTH-WEST
{
```

```
    //sjekk grensene
    while(y+mDistanse > bredde-1 || x-mDistanse < 0)
    {
        rehash();
    }

    if(mDistanse == 1)
    {
        //sett hinderet
        kart[x-1][y+1] = true;
    }
    else
    {
        int count = 1;
        while(count < mDistanse)
        {
            //fjern hinderet
            kart[x-count][y+count] = false;
            count++;
        }
        //sett hinderet
        if(mDistanse != 5)
        {
            kart[x-count][y+count] = true;
        }
    }

    //sett hindre bak
    while(y-lDistanse < 0 || x+lDistanse > bredde-1)
    {
        rehash();
    }

    if(lDistanse == 1)
    {
        //sett hinderet
        kart[x+1][y-1] = true;
    }
    else if(lDistanse == 2)
    {
        //slett hinder
        kart[x+1][y-1] = false;
        //sett hinder
        kart[x+2][y-2] = true;
    }
    else
    {
        //slett hindre
        kart[x+1][y-1] = false;
        kart[x+2][y-2] = false;
    }
}
else if(retning == VEST)//WEST
{
    //sjekk grensene
    while(x-mDistanse < 0)
    {
        rehash();
    }

    if(mDistanse == 1)
    {
```

```
            //sett hinderet
            kart[x-1][y] = true;
        }
        else
        {
            int count = 1;
            while(count < mDistanse)
            {
                //fjern hinderet
                kart[x-count][y] = false;
                count++;
            }
            //sett hinderet
            if(mDistanse != 5)
            {
                kart[x-count][y] = true;
            }
        }

        //sett hindre bak
        while(x+lDistanse > bredde-1)
        {
            rehash();
        }

        if(lDistanse == 1)
        {
            //sett hinderet
            kart[x+1][y] = true;
        }
        else if(lDistanse == 2)
        {
            //slett hinder
            kart[x+1][y] = false;
            //sett hinder
            kart[x+2][y] = true;
        }
        else
        {
            //slett hindre
            kart[x+1][y] = false;
            kart[x+2][y] = false;
        }
    }
    else //NORTH-WEST
    {
        //sjekk grensene
        while(x-mDistanse < 0 || y-mDistanse < 0)
        {
            rehash();
        }

        if(mDistanse == 1)
        {
            //sett hinderet
            kart[x-1][y-1] = true;
        }
        else
        {
            int count = 1;
            while(count < mDistanse)
            {
```

```
                //fjern hinderet
                kart[x-count][y-count] = false;
                count++;
            }
            //sett hinderet
            if(mDistanse != 5)
            {
                kart[x-count][y-count] = true;
            }
        }

        //sett hindre bak
        while(y+lDistanse > vidde-1 || x+lDistanse > bredde-1)
        {
            rehash();
        }

        if(lDistanse == 1)
        {
            //sett hinderet
            kart[x+1][y+1] = true;
        }
        else if(lDistanse == 2)
        {
            //slett hinder
            kart[x+1][y+1] = false;
            //sett hinder
            kart[x+2][y+2] = true;
        }
        else
        {
            //slett hindre
            kart[x+1][y+1] = false;
            kart[x+2][y+2] = false;
        }
    }
}

/**
 * Beveg roboten ved å endre x- og y-koordinatene i forhold
 * til retningen og hjulfarten. Må også sjekke om koordinatendringene
 * går utenfor kartet når den beveger seg bakover. Fremover går det
 * av seg selv, siden den ser fremover lenger enn den kan kjøre og
 * vil ekspandere kartet hvis den ser et hinder utenfor kartet.
 */
public void move(int cRetning, int hjulFart)
{
    retning = cRetning; //endre retning i forhold til kompassinnlesingen.
    int fart = hjulFart; //0: ingen bevegelse; 1: fremover bevegelse; 2: bakover bevegelse
    int fremover = 1;
    int bakover = 2;



    if(fart > 0)//farten til hjulene dikterer om den vil bevege seg på kartet.
    {
        if(retning == NORD) //NORTH
        {
            if(fart == fremover)
            {
                y--;
            }
```

```
    else
    {
        y++;
    }
}
else if(retning == NORD_ØST) //NORTH-EAST
{
    if(fart == fremover)
    {
        x++; y--;
    }
    else
    {
        x--; y++;
    }
}
else if(retning == ØST) //EAST
{
    if(fart == fremover)
    {
        x++;
    }
    else
    {
        x--;
    }
}
else if(retning == SØR_ØST) //SOUTH-EAST
{
    if(fart == fremover)
    {
        x++; y++;
    }
    else
    {
        x--; y--;
    }
}
else if(retning == SØR) //SOUTH
{
    if(fart == fremover)
    {
        y++;
    }
    else
    {
        y--;
    }
}
else if(retning == SØR_VEST) //SOUTH-WEST
{
    if(fart == fremover)
    {
        x--; y++;
    }
    else
    {
        x++; y--;
    }
}
else if(retning == VEST) //WEST
{
```

```
            if(fart == fremover)
            {
                x--;
            }
            else
            {
                x++;
            }
        }
        else //NORTH-WEST
        {
            if(fart == fremover)
            {
                x--; y--;
            }
            else
            {
                x++; y++;
            }
        }
    }
}

/**
 * Simuler bevegelsen av roboten, uten å skifte x- og y-
 * koordinater til roboten, og sjekk for hindringer fremover
 * og bakover. Denne skal også beregne en fitnessverdi ut fra
 * dette, slik at den kan brukes direkte i fitness-funksjonen.
 */
public int simuler(int cRetning, int hjulFart)
{
    retning = cRetning;
    int fart = hjulFart;//sjekkFart(hjulFart);
    int ySim = y;
    int xSim = x;
    //x- og y-koordinater fremover
    int y1 = 0;
    int x1 = 0;
    int y2 = 0;
    int x2 = 0;
    //x- og y-koordinater bakover
    int y3 = 0;
    int x3 = 0;
    int y4 = 0;
    int x4 = 0;

    //boolske variable for å sjekke hindre
    boolean enFrem = false;
    boolean toFrem = false;
    boolean enBak = false;
    boolean toBak = false;

    int distanseSim = 1;
    int fitnessSim = 0;

    int stopp = 0;
    int fremover = 1;
    int bakover = 2;
    int distanse = 1;

    //sjekk for grenseverdier på kartet
    if(ySim+2 >= vidde || ySim-2 < 0 || xSim+2 >= bredde || xSim-2 < 0)
```

```
{
    rehash();
}

//sjekk farten
if(fart > stopp)
{
    fitnessSim++;
}

//sjekk retning i forhold til hvor roboten er
if(retning == NORD)
{
    //endre koordinatene
    if(fart == fremover && !kart[xSim][ySim-1])
    {
        ySim--;
    }
    else if(fart == bakover && kart[xSim][ySim+1])
    {
        ySim++;
    }

    //sjekk for hindre
    //distanse fremover
    y1 = ySim-distanse;
    y2 = ySim-(distanse+1);
    enFrem = kart[xSim][y1];
    toFrem = kart[xSim][y2];
    if(!enFrem && !toFrem)
    {
        fitnessSim++;
    }

    //hinder bak
    y3 = ySim+distanse;
    y4 = ySim+distanse+1;
    enBak = kart[xSim][y3];
    toBak = kart[xSim][y4];
    if(!enBak && !toBak)
    {
        fitnessSim++;
    }
}
else if(retning == NORD_ØST)
{
    //endre koordinatene
    if(fart == fremover && !kart[xSim+1][ySim-1])
    {
        ySim--; xSim++;
    }
    else if(fart == bakover && !kart[xSim-1][ySim+1])
    {
        ySim++; xSim--;
    }

    //sjekk for hindre
    //distanse fremover
    y1 = ySim-distanse;
    y2 = ySim-(distanse+1);
    x1 = xSim+distanse;
    x2 = xSim+distanse+1;
```

```
        enFrem = kart[x1][y1];
        toFrem = kart[x2][y2];
        if (!enFrem && !toFrem)
        {
            fitnessSim++;
        }

        //hinder bak
        y3 = ySim+distanse;
        y4 = ySim+distanse+1;
        x3 = xSim-distanse;
        x4 = xSim-(distanse+1);
        enBak = kart[x3][y3];
        toBak = kart[x4][y4];
        if (!enBak && !toBak)
        {
            fitnessSim++;
        }
    }
    else if(retning == ØST)
    {
        //endre koordinatene
        if( fart == fremover && !kart[xSim+1][ySim])
        {
            xSim++;
        }
        else if( fart == bakover && !kart[xSim-1][ySim])
        {
            xSim--;
        }

        //sjekk for hindre
        //distanse fremover
        x1 = xSim+distanse;
        x2 = xSim+distanse+1;
        enFrem = kart[x1][ySim];
        toFrem = kart[x2][ySim];
        if (!enFrem && !toFrem)
        {
            fitnessSim++;
        }

        //hinder bak
        x3 = xSim-distanse;
        x4 = xSim-(distanse+1);
        enBak = kart[x3][y];
        toBak = kart[x4][y];
        if (!enBak && !toBak)
        {
            fitnessSim++;
        }
    }
    else if(retning == SØR_ØST)
    {
        //endre koordinatene
        if( fart == fremover && !kart[xSim+1][ySim+1])
        {
            xSim++; ySim++;
        }
        else if( fart == bakover && !kart[xSim-1][ySim-1])
        {
            xSim--; ySim--;
```

```
    }

    //sjekk for hindre
    //distanse fremover
    y1 = ySim+distanse;
    y2 = ySim+distanse+1;
    x1 = xSim+distanse;
    x2 = xSim+distanse+1;
    enFrem = kart[x1][y1];
    toFrem = kart[x2][y2];
    if (!enFrem && !toFrem)
    {
        fitnessSim++;
    }

    //hinder bak
    y3 = ySim-distanse;
    y4 = ySim-(distanse+1);
    x3 = xSim-distanse;
    x4 = xSim-(distanse+1);
    enBak = kart[x3][y3];
    toBak = kart[x4][y4];
    if (!enBak && !toBak)
    {
        fitnessSim++;
    }
}
else if(retning == SØR)
{
    //endre koordinatene
    if(fart == fremover && !kart[xSim][ySim+1])
    {
        ySim++;
    }
    else if(fart == bakover && !kart[xSim][ySim-1])
    {
        ySim--;
    }

    //sjekk for hindre
    //distanse fremover
    y1 = ySim+distanse;
    y2 = ySim+distanse+1;
    enFrem = kart[xSim][y1];
    toFrem = kart[xSim][y2];
    if (!enFrem && !toFrem)
    {
        fitnessSim++;
    }

    //hinder bak
    y3 = ySim-distanse;
    y4 = ySim-(distanse+1);
    enBak = kart[xSim][y3];
    toBak = kart[xSim][y4];
    if (!enBak && !toBak)
    {
        fitnessSim++;
    }
}
else if(retning == SØR_VEST)
{
```

```
        //endre koordinatene
        if(fart == fremover && !kart[xSim-1][ySim+1])
        {
            xSim--; ySim++;
        }
        else if(fart == bakover && !kart[xSim+1][ySim-1])
        {
            xSim++; ySim--;
        }

        //sjekk for hindre
        //distanse fremover
        y1 = ySim+distanse;
        y2 = ySim+distanse+1;
        x1 = xSim-distanse;
        x2 = xSim-(distanse+1);
        enFrem = kart[x1][y1];
        toFrem = kart[x2][y2];
        if(!enFrem && !toFrem)
        {
            fitnessSim++;
        }

        //hinder bak
        y3 = ySim-distanse;
        y4 = ySim-(distanse+1);
        x3 = xSim+distanse;
        x4 = xSim+distanse+1;
        enBak = kart[x3][y3];
        toBak = kart[x4][y4];
        if(!enBak && !toBak)
        {
            fitnessSim++;
        }
    }
    else if(retning == VEST)
    {
        //endre koordinatene
        if(fart == fremover && !kart[xSim-1][ySim])
        {
            xSim--;
        }
        else if(fart == bakover && !kart[xSim+1][ySim])
        {
            xSim++;
        }

        //sjekk for hindre
        //distanse fremover
        x1 = xSim-distanse;
        x2 = xSim-(distanse+1);
        enFrem = kart[x1][ySim];
        toFrem = kart[x2][ySim];
        if(!enFrem && !toFrem)
        {
            fitnessSim++;
        }

        //hinder bak
        x3 = xSim+distanse;
        x4 = xSim+distanse+1;
        enBak = kart[x3][ySim];
```

```
            toBak = kart[x4][ySim];
            if(!enBak && !toBak)
            {
                fitnessSim++;
            }
        }
        else if(retning == NORD_VEST)
        {
            //endre koordinatene
            if(fart == fremover && !kart[xSim-1][ySim-1])
            {
                xSim--; ySim--;
            }
            else if(fart == bakover && !kart[xSim+1][ySim+1])
            {
                xSim++; ySim++;
            }

            //sjekk for hindre
            //distanse fremover
            y1 = ySim-distanse;
            y2 = ySim-(distanse+1);
            x1 = xSim-distanse;
            x2 = xSim-(distanse+1);
            enFrem = kart[x1][y1];
            toFrem = kart[x2][y2];
            if(!enFrem && !toFrem)
            {
                fitnessSim++;
            }

            //hinder bak
            y3 = ySim+distanse;
            y4 = ySim+distanse+1;
            x3 = xSim+distanse;
            x4 = xSim+distanse+1;
            enBak = kart[x3][y3];
            toBak = kart[x4][y4];
            if(!enBak && !toBak)
            {
                fitnessSim++;
            }
        }
        return fitnessSim;
    }
}
```

Referanser

[1] V. Chvátal D. L. Applegate R. E. Bixby og W. J. Cook. *The Traveling Salesman Problem: A Computational Study*. Princeton University Press, 2006.

[2] Agoston E. Eiben og J. E. Smith. *Introduction to Evolutionary Computing*. SpringerVerlag, 2003. ISBN 3540401849.

[3] Dario Floreano og Francesco Mondada. Automatic creation of an autonomous agent: Genetic evolution of a neural-network driven robot. I *In*, side 421–430. MIT Press, 1994.

[4] A Kurz. Constructing maps for mobile robot navigation based on ultrasonic range data. *IEEE Trans. Systems, Man, and Cybernetics - Part B: Cybernetics*, (26):233–242, 1996.

[5] Jean arcady Meyer. Evolutionary approaches to neural control in mobile robots. I *In Proceedings of the IEEE International Conference on Systems, Man and Cybernetics*, side 1–43. Springer Verlag, 1998.

[6] David S. Johnson Michael R. Garey. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman, 1979.

[7] Julian Togelius og Simon M. Lucas. Evolving controllers for simulated car racing, 2005. Winner of the best student paper award at CEC 2005.

[8] John J. Watkins. *Across the Board: The Mathematics of Chessboard Problems*. Princeton University Press, 2004.