Genetiska algoritmer – hur påverkar dess parametrar evolutionen?

ANDERS SANDSTRÖM
och MARKUS ÖSTBERG

KTH Datavetenskap
och kommunikation

Examensarbete
Stockholm, Sverige 2011

Genetiska algoritmer – hur påverkar dess parametrar evolutionen?

A N D E R S S A N D S T R Ö M
o c h M A R K U S Ö S T B E R G

Examensarbete i datalogi om 15 högskolepoäng
vid Programmet för datateknik
Kungliga Tekniska Högskolan år 2011
Handledare på CSC var Johan Boye
Examinator var Mads Dam

URL: www.csc.kth.se/utbildning/kandidatexjobb/datateknik/2011/sandstrom_anders_OCH_ostberg_markus_K11006.pdf

Kungliga tekniska högskolan
Skolan för datavetenskap och kommunikation

KTH CSC
100 44 Stockholm

URL: www.kth.se/csc

Referat

Genetiska algoritmer är en metod för att lösa optimeringsproblem genom att efterlikna naturens evolution. I arbetet som leder fram till den här rapporten använder vi genetiska algoritmer för att lösa ett komplext problem. Problemet är att ta fram ett fordon som klarar av att ta sig igenom en på förhand definierad hinderbana. Huvuddelen av rapporten ägnas åt att undersöka vilka värden som är lämpliga att använda på de parametrar som finns i den genetiska algoritmen, så att den ska bli så effektiv som möjligt.

Abstract

Genetic algorithms
How does its parameters affect evolution?

Genetic algorithms is a method to solve optimization problems by imitating natural evolution. In the work that leads up to this report we use genetic algorithms to solve a complex problem. The problem is to develop a vehicle that can get through a predefined obstacle course. The main part of this report is devoted to examining which values that are suitable to use on the parameters that exist in the genetic algorithm, so that it will be as efficient as possible.

Förord

Den här rapporten är skriven som en del av kandidatexamensarbetet i kursen *Examensarbete inom datalogi* vid *Skolan för datavetenskap och kommunikation* (CSC) på *Kungliga Tekniska högskolan*. Rapporten är skriven av Anders Sandström och Markus Östberg. Arbetet med rapporten och den simulering, som användes för att testa den genetiska algoritmen, vi implementerat var uppdelat jämt mellan båda rapportförfattarna. Anders har haft huvudansvaret för att utveckla den grafiska simuleringen, medan Markus har ansvarat för kapitlen med introduktion och bakgrundsteori i rapporten. Utvecklingen och implementationen av den genetiska algoritmen samt undersökningen och skrivandet av diskussionen har vi tillsammans genomfört.

Vi vill tacka vår handledare Johan Boye för den excellenta hjälp han har givit oss under arbetet.

Innehåll

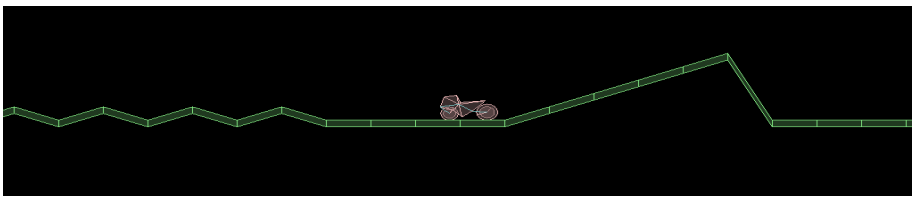
Kapitel 1

Introduktion

1.1 Inledning

Genetiska algoritmer är en heuristik för att lösa optimerings- och sökproblem. Genetiska algoritmer löser problem på ett sätt som är inspirerat av naturlig evolution.

Vi vill i detta projekt undersöka hur olika parametrar påverkar evolutionen hos den genetiska algoritmen, samt se hur bra genetiska algoritmer fungerar för att lösa komplexa problem. Detta genom att implementera en genetisk algoritm som tar fram ett fordon som ska klara av att ta sig fram på en i förhand definierad bana. Målet för algoritmen är att det fordon som genereras fram ska klara hela den fördefinierade banan. Den genetiska algoritmen kommer genom att optimera parametrarna för till exempel fordonets karossform, hjulstorlek, hjulens vridmoment och så vidare att ta fram ett fordon som klarar banan. Figur 1.1 visar ett fordon under evaluering på en del av banan.



Figur 1.1. Bild på ett fordon och en del av banan.

Därefter ska vi undersöka hur olika parametrar i den genetiska algoritmens påverkar utvecklingen av fordonet. Detta för att ta reda på om det finns någon optimal parameteruppställning för just det här problemet.

Denna rapport är ett examensarbete på grundnivå inom datalogi. Det innebär att det ställs höga krav på vetenskaplig noggrannhet samt att arbetet ska vara 'state of the art', vilket innebär att arbetet ska behandla det senaste tekniken inom det valda området.

1.2 Bakgrund

Evolutionära algoritmer som är förlagan till genetiska algoritmer, där man med datorers hjälp simulerade evolution, började först att utvecklas under mitten av 1950-talet av en matematiker vid namn Nils Aall Barricelli. [8] Under hans arbete med *Symbiogenetic Evolution Processes Realized by Artificial Methods* skapades och kördes bland de första enkla simuleringarna av artificiellt liv på datorer.

Från slutet av 50-talet började en forskare i genetik vid namn Alex Fraser att publicera ett antal avhandlingar och böcker om artificiell selektion, däribland *Computer Models in Genetics* som publicerades 1970. [8] Fraser och andra forskares arbete lade grund till att biologer i allt större utsträckning under slutet av 60-talet och början av 70-talet började använda datorer för att simulera evolution.

Genetiska algoritmer hade hittills inte varit ett stort forskningsområde, men genom John Hollands arbete i början av 1970-talet fick genetiska algoritmer allt större uppmärksamhet. Speciellt viktigt inverkan på detta anses Hollands bok *Adaptation in Natural and Artificial Systems* från 1975 ha haft. [8][2]

Fram till mitten av 1980-talet var forskningen inom genetiska algoritmer främst på ett teoretiskt plan. Det fanns helt enkelt inte datorer som var kraftfulla nog för att kunna genomföra de beräkningar som krävdes för mer avancerade genetiska algoritmer. När sedan datorutvecklingen hann ifatt teorin i slutet av 80-talet utvecklades det ett antal produkter som använde genetiska algoritmer för att optimera olika typer av problem.[8] 1989 släppte den första kommersiella programvaran som använde sig av genetiska algoritmer av företaget Axcelis, Inc. Deras program *Evolver* skulle kunna lösa ett stort antal problem genom att evolutionärt ta fram den bästa lösningen. [4]

1.3 Syfte

Syftet med denna rapport är att få en djupare förståelse om hur olika parametrar påverkar evolutionen hos en genetisk algoritm. Vi vill genom denna undersökning ta reda på vilken påverkan olika val för mutation, selektion och rekombination ger på evolutionen och om det finns någon kombination som kan tänkas vara optimal för just vårt problem. Detta genom att implementera en genetisk algoritm och genomföra ett antal tester där vi mäter prestandan hos algoritmen för olika variatoner hos de olika parametrarna.

Kapitel 2

Bakgrundsteori

2.1 Genetik och evolution

Här förklaras kort grundläggande begrepp inom genetik och evolution som läsaren bör vara förtrogen med för att kunna tillgodogöra sig innehållet i denna rapport:

Genom

Genomet utgör den information hos en organism som är ärftlig. Alltså i en människas fall den fullständiga DNA-sekvensen som vi har i en uppsättning kromosomer. Genomet utgörs av ett antal gener som var och en påverkar olika egenskaper hos organismen och som ärvs från generation till generation. [10]

Population

Population är en grupp individer av samma art som finns inom ett begränsat område under en viss tid. [10]

Mutation

Mutation är slumpmässiga förändringar av genomet. Mutation innebär ofta att en gen förändras slumpmässigt och att organismen därför får nya egenskaper som inte ärvts från någon förälder. [10]

Selektion

Selektion eller naturligt urval är det som avgör om en organism får fortplanta sig och på så sätt sprida sina gener till nästa generation. Selektion bestäms ofta till stor del av organismens lämplighet/fitness, men även till en del av slumpmässiga händelser. [10]

Fitness

Fitness eller lämplighet är den grad av anpassning som en individ har till sin miljö. Hög fitness innebär att individen har en stor sannolikhet att reproducera sig och sprida sina gener vidare. [10]

Rekombination

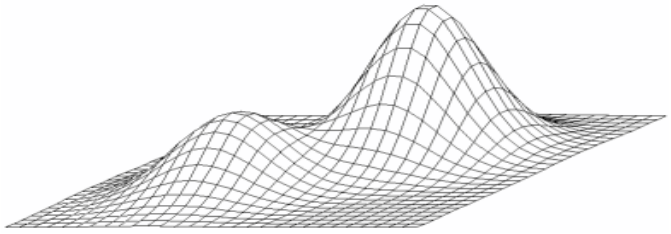
Rekombination innebär att en ny organism bildas genom att man kombinerar ihop gener från organismens föräldrar. [10]

2.2 Optimerings- och sökproblem

Att optimera något är en process där man justerar ett antal variabler för en apparat, ett matematiskt problem eller en situation för att hitta den bästa lösningen. [2] Det kan handla om att hitta den snabbaste vägen mellan två platser i en stad eller att bygga en bil som drar så lite bränsle som möjligt. Variablerna som man justerar kan vara allt från vilken väg man väljer att ta eller vilken tid på dygnet man gör resan, till vinkeln på vindrutan eller bredden på bilen.

Det finns många olika sätt att lösa optimeringsproblem. Enklare optimeringsproblem kan man ofta lösas matematiskt genom att beräkna derivatan och söka dess nollställe, men detta är inte alltid möjligt.

Ett exempel på ett svårare optimeringsproblem är *Handelsresandeproblemet*. Problemet som går ut på att hitta kortaste vägen som passerar ett antal städer kan uttryckas som att hitta maximum eller minimum för någon värdefunktion. Detta kan illustreras som att hitta maximumpunkten i ett tredimensionellt rum. Se Figur 2.1.



Figur 2.1. Visar en yta med varierande höjd. Höjden symboliserar värdefunktionen. [11]

Den vanliga metoden för att lösa ett sådant problem är att använda sig av en så kallad *Hill climbing* algoritm. Dessa fungerar i stor sätt så att man går i den riktning som det lutar brantast i tills dess att man hittar en topp. Problemet med detta är att man inte kan garantera att det är det globala maximumet man hittat. Detta då man, beroende på startposition, lika gärna skulle kunnat fastnat i ett lokalt maximum. Figur 2.1 är ett exempel på situation där en hill climbing algoritm skulle kunna ge fel svar. [11]

2.3 Genetiska algoritmer

2.3.1 Allmänt om genetiska algoritmer

En genetisk algoritm arbetar genom att den har en samling individer, den så kallade populationen. När man startar så är individerna i populationen oftast slumpartat skapade, men de kan också ha satts till flera kopior av en individ som man tagit fram innan och nu vill optimera.

Algoritmen utvärderar sedan lämpligheten på alla individer i populationen, detta steg är applikationsspecifikt och går i vår undersökning ut på att köra individen i vår fysiksimulering och se hur långt den kan komma. När man vet hur lämplig varje individ är kan man göra en selektion i populationen där man väljer ut de individer som ska bli föräldrar till nästa generation. Dessa individer korsas sedan med ett visst antal av de andra selekterade individerna. Hur många andra de korsas med beror på utformandet av rekombinationsalgoritmen. Därefter muteras slutligen de rekombinerade individerna för att möjliggöra uppkomsten av nya egenskaper.

Man har då skapat nästa generation och algoritmen börjar om med att utvärdera lämpligheten på populationen. När man kommit fram till en individ med tillräckligt hög lämplighet eller att nya individer med högre lämplighet inte utvecklats fram avslutar man den genetiska algoritmen.[7]

2.3.2 Varför använda genetiska algoritmer?

Om vi återgår till problemet att hitta den högsta punkten i ett koordinatsystem så är genetiska algoritmer en bättre metod än hill climbing då en bra kodad genetisk algoritm inte skulle fastna i det lokala maximumet. [2]

Denna egenskap hos genetiska algoritmer gör att de är ett utmärkt val när man ska hitta en optimal lösning på ett komplext problem, förutsatt att evalueringen av hur bra lösningen är inte tar allt för lång tid [2]. Det är dock inte garanterat att man hittar den optimala lösningen då genetiska algoritmer är en heuristik, vilket innebär att man inte kan garantera att en optimal lösning hittas.

2.3.3 Genomdesign

Det finns två vanliga sätt att spara genomet till genetiska algoritmer på. Det vanligaste sättet är att lagra genomet som en binär sträng där man kodar alla parametrarna binärt och sedan lagrar dem efter varandra i en sträng. Den här typen av kodning är ofta inte naturlig för många problem, man får också vissa komplikationer. Till exempel kan variabeln som en gen motsvarar ligga utanför dess gränsvärden efter mutation eller rekombination och man måste då korrigera det i efterhand.

En alternativ metod är att istället implementera problemets variabler i dess naturliga kodning, exempelvis flyttal. Denna metod kallas för att man lagrar genomet värdekodat.

I tabell 2.1 visas hur värden kan lagras på de två olika sätten för att representera genomet.

Tabell 2.1. Olika metoder att lagra genomet.

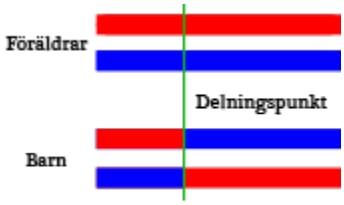
Binär sträng:	"10101101011101111001101110101010110001010011000100110010011100111100"
Värdekodat:	{ 1.3552, 2.4323, 4.3245, 1.1532 }

Det finns fördelar och nackdelar med båda metoderna. Fördelen med att implementera genomet värdekodat är att man kan representera gener mer problemnära. Exempelvis kan man ha flyttal för en egenskap som man uttrycker med ett reelt tal. En nackdel är dock att man då bara kan förlita sig på mutationen för att få nytt genetiskt material. Med en bitsträng kan även rekombinationen ge upphov till nytt genetiskt material då rekombinationen kan kombinera bitar som utgör en parameter hos varje förälder till ett helt nytt värde i dess avkomma. [2]

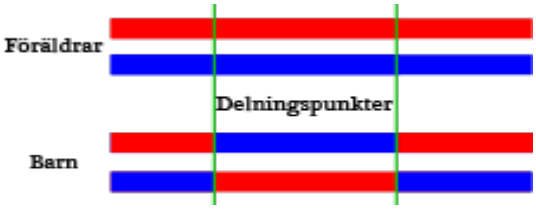
2.3.4 Rekombination

Rekombination är hur generna från föräldraindividerna kombineras ihop för att skapa en ny individ. Beroende på hur man kombinerar ihop de olika föräldrarnas genetiska material så kommer den nya individen att kunna få vitt skilda egenskaper. [9]

Den vanligaste och enklaste metoden för rekombination är att välja ett eller två delningspunkter för att sedan dela upp och kombinera om genomet från de två föräldrarna till två barn. Figurerna 2.2 och 2.3 illustrerar hur rekombinationen fungerar om man har en eller två delningspunkter.



Figur 2.2. Rekombination med en delningspunkt. [9]



Figur 2.3. Rekombination med två delningspunkter. [9]

2.3.5 Selektion

Selektion är hur man väljer vilka individers genom som ska få leva vidare till nästa generation. Selektion kan göras på flera sätt. Vanligast är turneringsselektion och rouletthjulsselektion. Man kan även kombinera dessa selektionssätt med elitselektion.

Turneringsselektion fungerar så att man sätter upp en turnering för en begränsad del av populationen. De individer med högst fitness, av de i turneringen, väljs för att rekombineras till nästa generation. Detta upprepas tills dess att kvoten för antalet individer som ska gå vidare till rekombination är fylld. Beroende på hur stor denna kvot är och hur många individer som ingår i varje turnering så kan man ändra selektionstrycket på populationen. Om till exempel alla individer i en population skulle vara i samma turnering så skulle enbart de med högst fitness gå vidare till nästa generation, alltså har vi då ett högt selektionstryck. På samma sätt skulle turneringar med endast två individer leda till ett lågt selektionstryck då även individer med sämre fitness skulle ha en chans att gå vidare. [2]

Rouletthjulsselektion innebär att ha ett rouletthjul med exempelvis 100 fack där varje fack motsvarar att en individ går vidare. Genom att ge individer med hög fitness fler fack så kommer sannolikheten att de går vidare att vara högre. Har vi till exempel tre individer med fitness 10, 20 och 70 så kommer individen med fitness 70 att ha 70 st fack på rouletthjulet, individen med 20 i fitness 20 st fack och så vidare. [7]

Att kombinera någon av dessa med elitselektion innebär att man, innan man använder den andra selektionsalgoritmen, väljer att direkt ta med ett fåtal av de individer med absolut högst fitness. Detta för att försäkra sig om att dessa individer får sprida sina egenskaper till nästa generation.

2.3.6 Mutation

Mutation används av genetiska algoritmer för att behålla genetisk mångfald i populationen. Utan mångfald finns det en större risk att populationen fastnar på ett lokalt maximum av möjliga lösningar och inte utvecklas vidare till en bättre lösning.[2]

Mutation kan implementeras på flera sätt, det vanligaste är att man har en viss sannolikhet att en bit i genomet ska muteras. Vid mutationen inverteras då den biten. Den här metoden är enkel och liknar naturens mutation men kan tyvärr inte användas om man har ett värdekodat genom med flyttal.

Om man använder värdekodade genom så blir implementationen av mutationen problemspecifik, då den måste vara anpassad för de datatyper man använder i genomet.

2.3.7 Parametrar för genetiska algoritmer

I genetiska algoritmer finns det ett antal parametrar som påverkar hur en lösning till problemet utvecklar sig. Många av dessa parametrar är inspirerade av de parametrar som påverkar evolution så som populationsstorlek, arv, mutationshastighet, selektionstryck och rekombination. För att förstå hur genetiska algoritmer fungerar är det därför viktigt att ha kunskap om hur evolution fungerar. [3]

Populationsstorlek

Populationsstorleken är antal individer i varje generation. En stor population ger stor genetisk mångfald, men algoritmen konvergerar då långsammare. Med en liten populationsstorlek så konvergerar algoritmen snabbare. Dock så sparas då inte så många olika lösningsvarianter och det händer lättare att nya varianter som har stor potential försvinner till nästa generation, detta kallas för genetisk drift. [2]

Selektionstryck

Selektionstryck är den parameter som hänger ihop med selektion, med ett högt selektionstryck väljs bara de bästa individerna ut till nästa generation. Med ett högt selektionstryck kan man få samma problem som med en låg mutationshastighet, populationen fastnar i ett lokalt maximum och utvecklas inte vidare. Om man istället har ett lågt selektionstryck konvergerar den genetiska algoritmen onödigt långsamt mot den optimala lösningen. [6] Selektionstrycket beror på parametrar som är specifika för den algoritmen som valts för selektion. En parameter som finns i alla vanliga selektionsfunktioner är selektionsstorleken. Den avgör hur stor del av populationen som går vidare till rekombination, och påverkar därför selektionstrycket.

Mutationssannolikheten

Mutationssannolikheten är den parameter som hänger ihop med mutation, den beskriver sannolikheten att en viss bit av den genetiska sekvensen ska ändras till nästa generation. [2]

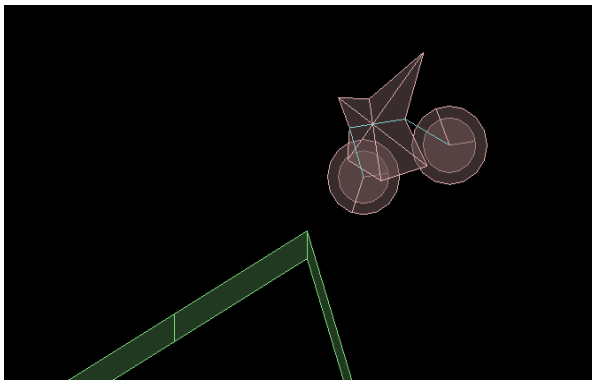
Kapitel 3

Implementationen

3.1 Simuleringen

3.1.1 Fordonet

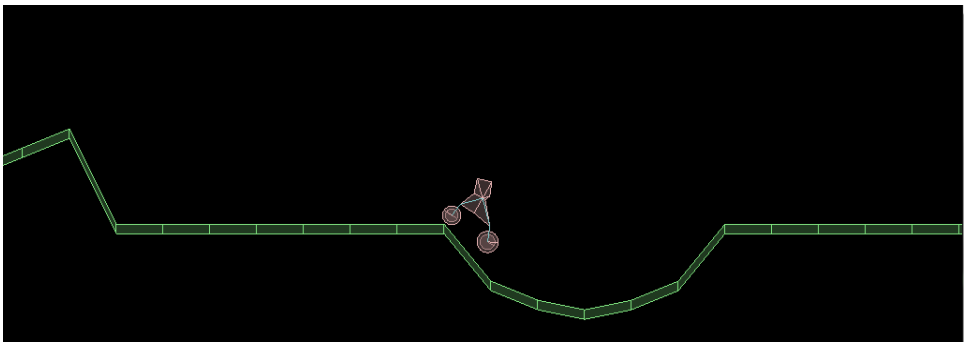
Fordonet består av en kaross och två stycken hjul. Karossen byggs upp av 8 stycken kanter som kan ha varierande längd. Hjulen sitter fast i en stötdämpare som kan röra sig in och ut i samma vinkel som den har mot fordonet. Detta gör att vinkeln mellan karossen och stötdämparen kan ha en avgörande roll för fordonets fitness. Figur 3.1 visar ett fordon som klarat hela evalueringsbanan och där man tydligt ser de kanter av olika längd som bygger upp karossen.



Figur 3.1. Visar ett fordon där kanterna som bygger upp karossen har varierande längd.

3.1.2 Evalueringsbanan

Evalueringsbanan består av ett antal block som sitter ihop. Den byggs upp av en vektor som innehåller y-värdena för varje skarvpunkt mellan två block. Skillnaden mellan x-värdena för varje block är alltid lika stort, vilket syns i figur 3.2.



Figur 3.2. Visar en sektion av evalueringsbanan och visar hur de olika blocken bygger upp banan.

3.1.3 Fysik och grafik

Till våra simuleringar behöver vi en fysikmotor, vi har valt att använda oss av en som heter Box2d [1], vilket är en fysikmotor för 2d objekt som är öppen källkod. Box2d har även OpenGL kod för att rita ut objekt för felsökning som vi använder för att lättare få en överblick av vad som händer under våra tester.

3.1.4 Slumpgeneratorn

Eftersom många delar av den genetiska algoritmen använder slump så är det viktigt att ha en bra slumpgenerator så att slumpgeneratorn inte är en avgörande felkälla i våra tester. Vi har valt att använda en implementation av *Mersenne twister* från c++ biblioteket *Boost* [5] som heter *mt19937*.

Mersenne twister är en såkallad *pseudorandom number generator* och det innebär att man ger algoritmen en *seed* innan man börjar hämta slumptal från den. Samma *seed* ger alltid samma slumptal och man kan därför återskapa en testomgång om man har kvar *seeden* från denna. Detta kan vara användbart vid till exempel felsökning eller om man vill återskapa en simulation.

3.2 Genetiska algoritmen

3.2.1 Genom

I vår implementation av den genetiska algoritmen har vi valt att implementera genomet som en vektor med alla de parametrar som behövs för att bygga upp fordonet. Detta eftersom det ger högre noggrannhet på parametrarna vilket är viktigt i för att kunna hitta en lösning på det problem vi implementerat den genetiska algoritmen för att lösa.

Gener för punkterna som fordonets kaross består av:

- Avstånd (A)- Bestämmer punktens avstånd från mittpunkten av karossen.

Gener för ett av fordonets hjul:

- Maxhastighet (H) - Bestämmer hur fort hjulet får snurra.
- Vridmoment (M) - Bestämmer med vilken kraft hjulet roterar.
- Storlek (S) - Bestämmer storleken på ett hjul.
- Position (P) - Bestämmer vilken punkt på fordonets kaross som hjulaxeln ska vara ansluten till.
- Vinkel (V)- Bestämmer vinkeln för hjulaxeln.

Dessa gener sitter sedan ihop i den struktur som visas i tabell 3.1 för att bilda fordonets genom:

Tabell 3.1. Struktur för genomet.

A1	A2	...	A8	H1:V	H1:P	H1:S	H1:M	H1:H	H2:V	H2:P	H2:S	H2:M	H2:H

Se ovan för förkortningarna. I tabell 3.1 motsvarar A1 till A8 längderna på de 8 st kanterna som bygger upp karossen. H1 är parametrarna för hjul 1 och H2 är parametrarna för hjul 2.

3.2.2 Rekombination

Vi har implementerat både rekombination med en delningspunkt och med två delningspunkter för att undersöka om det är någon avgörande skillnad mellan de båda. Algoritm 3.1 visar pseudokod för vår implementation av rekombination med två delningspunkter. Då rekombination med en delningspunkt endast är ett specialfall av rekombination med två delningspunkter, där man alltid har en delningspunkt i vektorns slut, så används i stort sett samma algoritm till den.

Algoritm 3.1. Pseudokod för rekombination med två delningspunkter. Rekombination med en delningspunkt är samma algoritm men med split2 satt till genomlängden -1.

```
in : vehicles
toMutation = {}
for i in range(0, vehicles.size)
    a = vehicles[i].genome
    b = vehicles[(i+1)%vehicles.size].genome

    split1 = random in range(1, genomeSize -1)
    split2 = random in range(1, genomeSize -1)
    if(split1 > split2)
        swap(split1, split2)

    swap(a[split1:split2], b[split1:split2])
    toMutation.add(a)
    toMutation.add(b)
return toMutation
```

3.2.3 Selektion

I vår implementation har vi valt att använda turneringsselektion då den är enkel att implementera och man kan enkelt justera selektionstrycket genom att modifiera turneringsstorleken. Algoritm 3.2 visar pseudokod för vår implementation av turneringsselektion.

Algoritm 3.2. Pseudokod för turneringsselektion.

```
in : vehicles
toCrossover = {}
for i in range(0, toCrossoverSize)
    tournament = {}

    for j in range (0, min(tournamentSize, vehicles.size))
        v = random vehicle in vehicles
        tournament.add(v)
        vehicles.remove(v)

    best = highest fitness in tournament
    toCrossover.add(best)
    tournament.remove(best)

    for v in tournament
        vehicles.add(v)
return toCrossover
```

3.2.4 Mutation

Eftersom vi har värdekodning på vårt genom så kan man inte använda det typiska sättet att genomföra mutation, att man ändrar en bit i genomsträngen med en viss sannolikhet. Vi har valt att använda en liknande variant där varje flyttal i vår vektor har en viss chans att mutera. Om ett flyttal muteras så slumpas det ett helt nytt värde som ligger inom de gränsvärden som gäller för just den typen av gen. Algoritm 3.3 visar pseudokod för vår implementation av mutationsalgoritmen.

Algoritm 3.3. Pseudokod för mutationsalgoritmen.

```
in : vehicles
for v in vehicles
    for g in genome
        if random in range(0, 1) < mutationrate
            g = random in range(g.low_limit, g.high_limit)
return vehicles
```

3.2.5 Fitness

För att kunna evaluera lämpligheten hos de individer som den genetiska algoritmen skapar så använder vi oss av fysikmotorn. Fitnessvärdet individen får är bilens position i x-led när körningen tar slut. Körningen kan antingen ta slut när individen har klarat hela banan eller om individen fastnar och inte kommer vidare. För att kontrollera om individen har fastnat eller inte har vi en funktion som undersöker om den har kommit en bit framåt under senaste tiden, om den inte har det så avbryter den körningen.

Kapitel 4

Undersökning

4.1 Utförande

Genom att köra vår implementation av den genetiska algoritmen ett antal gånger med olika parametrar, och undersöka hur många generationer som krävs för att den ska komma fram till ett fordon som klarar av hela den fördefinierade banan, så kan vi undersöka hur de olika parametrarna påverkar algoritmens effektivitet.

De standardvärden vi har använd i testerna är:

- En delningspunkt.
- 3% mutationssannolikhet.
- 40 i populationsstorlek.
- 2 i turneringsstorlek.

Om datan i testerna visar så små skillnader så man inte kan se att parametern vi ändrade på har effekt och om man inte heller kan se någon trend i datan så kommer vi göra ett så kallat t-test. Man får då ut en sannolikhet att testerna kommer från samma normalfördelning. Om datan kommer från samma normalfördelning har den parametern vi ändrat på ej påverkat algoritmens effektivitet.

4.1.1 Rekombination med en eller två delningspunkter

För att testa om det var någon större skillnad mellan rekombination med en eller två delningspunkter så valde vi att köra två tester, ett test för vardera typ av rekombinationsmetod. Testena användes för att undersöka hur många generationer som de olika rekombinationsalgoritmerna i snitt behövde innan den genetiska algoritmen fann en lösning.

Hypotesen var att rekombination med två delningspunkter skulle vara lite bättre då den rekombinationsmetoden borde ge större variation.

4.1.2 Mutationssannolikhet

För att testa vilken påverkan mutationssannolikheten har på algoritmen valde vi att köra 11 st tester med olika värden. De värden vi valde att testa med som mutationssannolikhet var: 0%, 1%, 2%, 3%, 4%, 5%, 6%, 7%, 8%, 9% och 10%.

Vår hypotes för testet med 0% var att det inte skulle kunna genomföras utan att det skulle stöta på problem med för tidig konvergens. Om 10% skulle gett bättre resultat än de andra så skulle vi kört fler tester på med värden över 10 % för att se om det finns ett bättre värde.

4.1.3 Turneringsstorlek

För att testa turneringsstorlekens påverkan på den genetiska algoritmen så valde vi att köra 5 st olika tester med följande värde på variabeln för turneringsstorlek: 2, 4, 8, 16 och 32.

4.1.4 Populationsstorlek

För att testa hur populationsstorleken påverkar valde vi att köra 6 st olika tester. De värden vi testade med var 10, 20, 40, 80, 160 och 320.

Då vi testar olika populationsstorlek så kan vi inte jämföra resultaten på samma sätt som i de andra testerna eftersom man hittar en lösning snabbare, i antal generationer, med större populationsstorlek. Tiden det tar att köra en generation ökar dock linjärt med populationsstorleken så man kan därför titta på hur många olika fordon som totalt måste evalueras innan en lösning är funnen. Därför måste vi multiplicera våra resultat på medelvärde och standardavvikelse med populationsstorleken på varje test. Man kan då se vilket värde på populationsstorleken som är effektivast.

4.2 Resultat

4.2.1 Rekombination, en eller två delningspunkter

Tabell 4.1 visar medelvärdet och standardavvikelsen för hur många generationer algoritmen i snitt behövde köras för att hitta en lösning. Ett lägre värde innebär att algoritmen i genomsnitt hittat en lösning snabbare än ett högre värde.

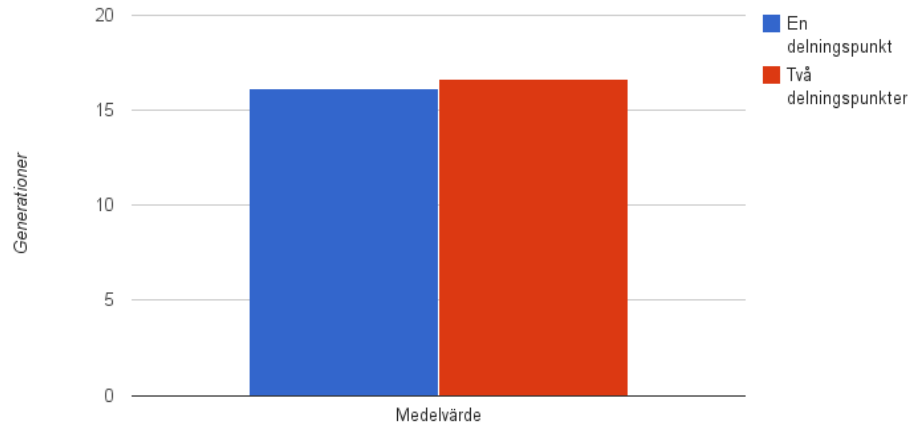
Tabell 4.1. Medelvärdet och standardavvikelsen för antal generationer innan en lösning var funnen.

Rekombinationsmetod	Medelvärde	Standardavvikelse	Stickprovsstorlek
En delningspunkt	16.15	10.16	515
Två delningspunkter	16.62	11.42	539

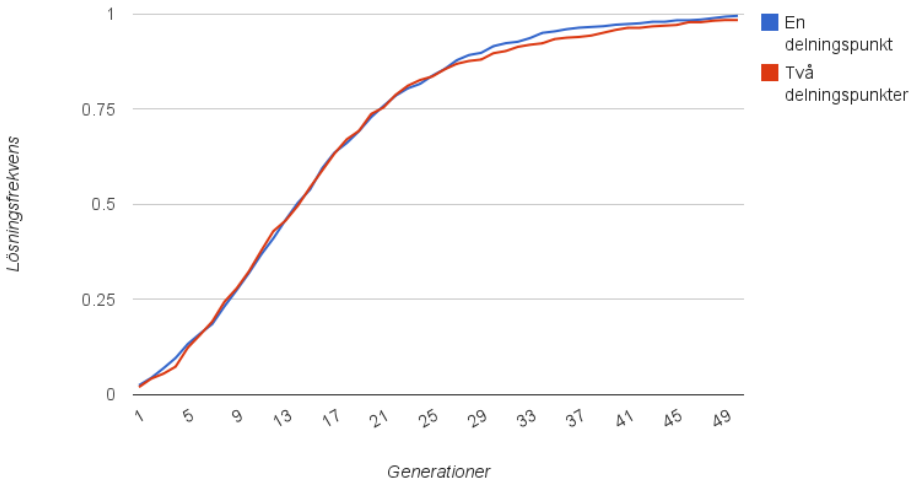
Då resultatet för båda dessa mätningar är mycket lika så gjorde vi ett T-test på de två grupperna av mätvärden. T-testet gav följande resultat: 0.473, vilket innebär att det med 47,3% sannolikhet inte är någon statistisk skillnad på medelvärdet för de två mängderna med stickprover.

Figur 4.1 visar medelvärdet för hur många generationer det krävdes vid de olika körningarna innan ett fordon som överlevde hela evalueringsbanan hade hittats. Varje stapel representerar medelvärdet från alla de körningar som gjordes med en viss parameter.

Figur 4.2 visar andelen av körningar som hittat en lösning efter ett visst antal generationer. Den avläses så att 25 % av körningarna, för både en och två delningspunkter, hade hittat en lösning efter ca 9 generationer.



Figur 4.1. Medelvärdet för hur många generationer som krävs innan algoritmen funnit ett svar med vid användning av de olika rekombinationsmetoderna. Ett lägre värde innebär att algoritmen i genomsnitt hittat en lösning snabbare än vid ett högt värde.



Figur 4.2. Andel körningar som funnit ett svar efter ett visst antal generationer med vid användning av de olika rekombinationsmetoderna.

4.2.2 Mutationssannolikhet

Tabell 4.2 visar medelvärdet och standardavvikelsen för hur många generationer algoritmen i snitt behövde köras för att hitta en lösning. Ett lägre värde innebär att algoritmen i genomsnitt hittat en lösning snabbare än ett högre värde.

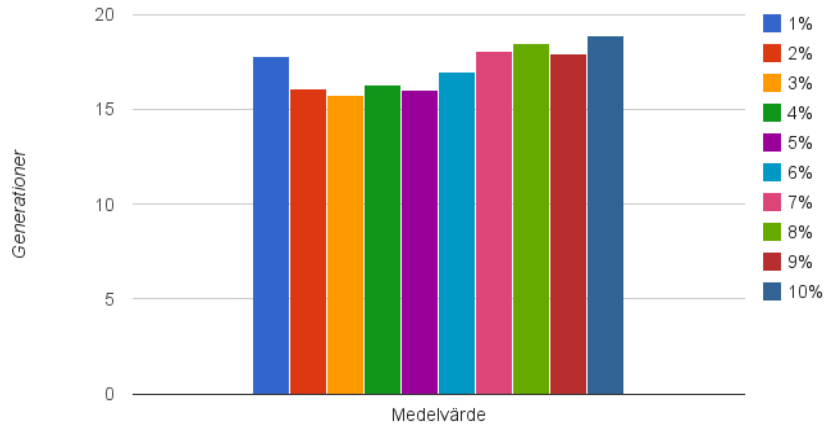
Testfallet med 0% mutationssannolikhet gav inget resultat då dessa körningar oftast fastnade med ett fordon som inte klarade av hela banan. Därför redovisas inte detta i tabellen eller figurerna.

Tabell 4.2. Medelvärdet och standardavvikelsen för antal generationer innan en lösning var funnen.

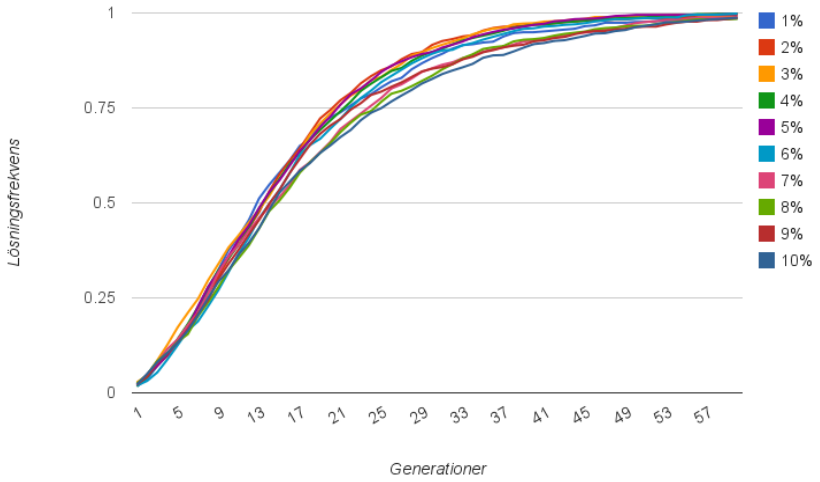
Mutationssannolikhet	Medelvärde	Standardavvikelse	Stickprovsstorlek
1%	17.79	21.66	441
2%	16.09	11.79	584
3%	15.70	10.91	989
4%	16.27	11.5	1012
5%	16.04	11.05	1070
6%	16.95	11.23	670
7%	18.08	14.08	639
8%	18.47	13.61	654
9%	17.89	13.60	695
10%	18.87	14.31	1079

Figur 4.3 visar medelvärdet för hur många generationer det krävdes vid de olika körningarna innan ett fordon som överlevde hela evalueringsbanan hade hittats. Varje stapel representerar medelvärdet från alla de körningar som gjordes med en viss parameter.

Figur 4.4 visar andelen av körningar som hittat en lösning efter ett visst antal generationer. Varje linje motsvarar hur körningarna med en viss mutationssannolikhet konvergerar mot att alla körningar hittat en lösning. Man ser till exempel att den genetiska algoritmen med en mutationssannolikhet på 10 % konvergerar långsammare än den med en mutationssannolikhet på 3 %.



Figur 4.3. Medelvärdet för hur många generationer som krävs innan algoritmen funnit ett svar med olika värden på mutationssannolikheten. Ett lägre värde innebär att algoritmen i genomsnitt hittat en lösning snabbare än vid ett högt värde.



Figur 4.4. Andel körningar som funnit ett svar efter ett visst antal generationer med olika värden på mutationssannolikheten.

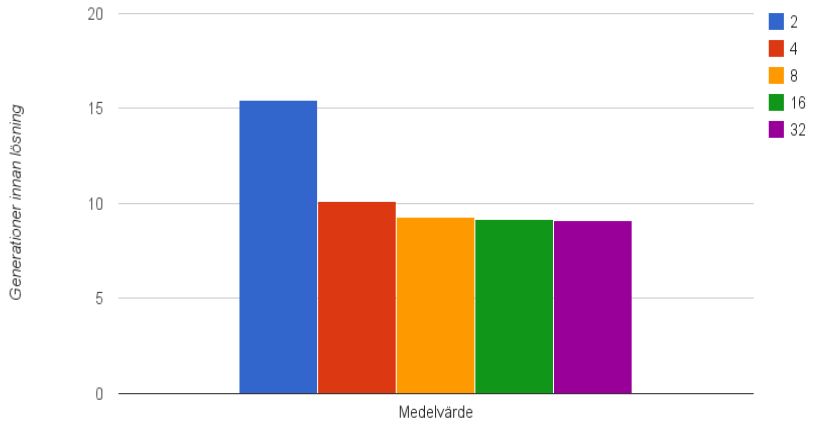
4.2.3 Turneringsstorlek

Tabell 4.3 visar medelvärdet och standardavvikelsen för hur många generationer algoritmen i snitt behövde köras för att hitta en lösning. Ett lägre värde innebär att algoritmen i genomsnitt hittat en lösning snabbare än ett högre värde.

Tabell 4.3. Medelvärdet och standardavvikelsen för antal generationer innan en lösning var funnen.

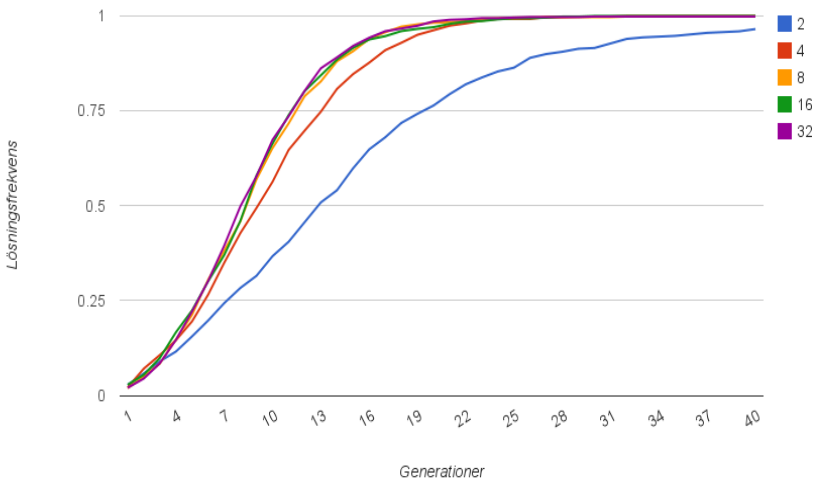
Turneringsstorlek	Medelvärde	Standardavvikelse	Stickprovsstorlek
2	15,39	11,26	502
4	10,07	5,31	666
8	9,28	4,95	678
16	9,16	4,74	678
32	9,06	4,58	696

Figur 4.5 visar medelvärdet för hur många generationer det krävdes vid de olika körningarna innan ett fordon som överlevde hela evalueringsbanan hade hittats. Varje stapel representerar medelvärdet från alla de körningar som gjordes med en viss parameter.



Figur 4.5. Medelvärdet för hur många generationer som krävs innan algoritmen funnit ett svar. Ett lägre värde innebär att algoritmen i genomsnitt hittat en lösning snabbare än ett högre värde.

Figur 4.6 visar andelen av körningar som hittat en lösning efter ett visst antal generationer. Varje linje motsvarar hur körningarna med en viss turneringsstorlek konvergerar mot att alla körningar hittat en lösning. Man ser till exempel att den genetiska algoritmen med en turneringsstorlek på 2 individer konvergerar långsammare än den med en turneringsstorlek på 32 individer.



Figur 4.6. Andel körningar som funnit ett svar efter ett visst antal generationer.

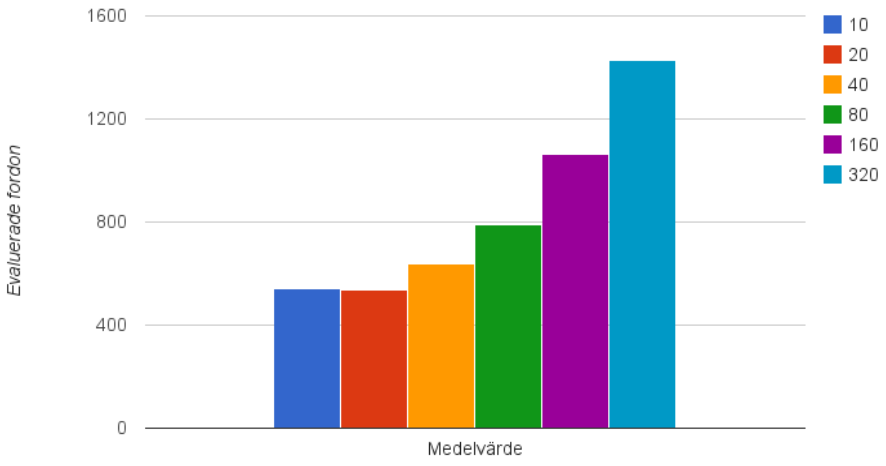
4.2.4 Populationsstorlek

Tabell 4.4 visar medelvärdet av antal evaluerade fordon innan algoritmen funnit en lösning. Ett lägre värde innebär att algoritmen i genomsnitt hittat en lösning snabbare än vid ett högt värde.

Tabell 4.4. Medelvärdet och standardavvikelsen för antal fordon som evaluerats innan en lösning var funnen.

Populationsstorlek	Medelvärde	Standardavvikelse	Stickprovsstorlek
10	539.66	558.63	594
20	533.55	417.97	766
40	633.16	434.67	754
80	788.46	511.64	728
160	1061.46	660.38	604
320	1425.87	828.38	487

Figur 4.7 visar medelvärdet för antal fordon som evaluerats vid de olika körningarna innan ett fordon som överlevde hela evalueringsbanan hade hittats.



Figur 4.7. Medelvärdet för hur många fordon som måste evalueras innan algoritmen funnit ett svar. Ett lägre värde innebär att algoritmen i genomsnitt hittat en lösning snabbare än vid ett högt värde.

Kapitel 5

Diskussion

5.1 Rekombination, en eller två delningspunkter

Resultatet från de två testerna med rekombination ger inget avgörande resultat i frågan om vilken av metoderna som är effektivast. Det finns en liten skillnad mellan de två, men dessa skulle lika väl kunna bero på statistiska avvikelser.

Då resultaten var så pass otydliga så beräknades T-test för de två resultaten. T-testet gav resultatet 0.473 vilket, som tidigare nämnts, innebär att det med 47,3% sannolikhet inte är någon statistisk skillnad på medelvärdet för de två mängderna med stickprover. Detta innebär alltså att det med stor sannolikhet inte är någon skillnad mellan att använda rekombination med en eller två delningspunkter.

Om det trots vårt resultat skulle vara någon skillnad mellan de två metoderna så är den för liten för att ge någon reell skillnad på effektiviteten hos vår implementation av den genetiska algoritmen.

5.2 Mutationssannolikhet

Testet för de olika mutationssannolikheterna har visat att det spelar en stor roll för konvergenshastigheten vilken mutationssannolikhet man använder. De som tydligt ger sämst resultat är mutationssannolikhet på 1% och alla de från 7% och upp till 10%. Variationen mellan testerna med 2%, 3%, 4%, 5 % och 6 % är så pass små att det inte går att utesluta om de beror på någon reell skillnad mellan de olika mutationssannolikheterna eller om det endast är statistiska avvikelser.

Det är dock tydligt att ett allt för lågt eller allt för högt värde på mutationssannolikheten har en negativ effekt på hur många generationer som måste evalueras innan den genetiska algoritmen finner en lösning. Utifrån våra tester är det troligt att det optimala värdet på mutationssannolikheten, för vårt problem, ligger någonstans mellan två och sex procent. Om man studerar trenden för medelvärdena så är det troligt att det optimala värdet för att lösa vårat problem ligger omkring 3%.

Anledningen till att 0% inte alls fungerar vid vissa körningar tror vi är att det i de körningarna antingen inte fanns, eller selekterades bort, gener som skulle

kunna utgöra en lösning. Vilket innebär att algoritmen till slut fastnar i ett lokalt maximum. Detta visar tydligt på vikten av att använda en bra mutationsalgoritm med ett välavpassat värde på mutationssannolikheten.

5.3 Turneringsstorlek

Testet för olika turneringsstorlekar visade att det var stora skillnader mellan en storlek på 2 och högre storlekar. Storlek 2 hade över 50% högre medelvärde i testdatan så den är utom rimligt tvivel sämre att använda i simuleringen. Man kan se en trend att ju högre turneringsstorlek desto lägre medelvärde. Man kan dock inte vara säker på om detta stämmer mellan storlekarna 8, 16 och 32 då skillnaderna mycket väl kan bero på statistiska avvikelser då de skiljer sig så pass lite.

Eftersom högre turneringsstorlek ger högre selektionstryck, vilket ger snabbare konvergens, så är det sannolikt att en turneringsstorlek på 32 skapar en lösning fortast. Ett högre selektionstryck kan även ge för snabb konvergens men det kan inte ses i datan vi har fått ut. Det är möjligt att man kan upptäcka detta om man analyserar mer än medelvärde och standardavvikelse på datan. Det kan också vara så att simuleringen inte har några tillräckligt kraftigt lokala maximum som den kan kan fastna i om den konvergerar för tidigt.

5.4 Populationsstorlek

I testet av olika populationsstorlekar fick man ett tydligt resultat att de större populationsstorlekarna inte är lika effektiva som de mindre. Storlek 10 och 20 gav väldigt lika resultat gällande medelvärde, men med storlek 20 var resultaten inte lika spridda eftersom 20 hade en lägre standardavikelse. Att de lägre populationsstorlekarna är effektivare för just den här simuleringen kan bero på att problemet är så pass lätt att den inte behöver mångfalden i individerna som den får med en större population. Om problemet hade varit svårare hade algoritmen haft svårare att hitta en lösning med en så pass liten population.

5.5 Sammanfattning

Resultaten i testerna med en eller två delningspunkter visade så små skillnader så det inte är möjligt att säga om de skiljer sig åt i effektivitet. Testerna med olika mutationssannolikheter visade att en mutationssannolikhet på 1% var för lågt då testerna med 2% till 6% uppvisade bättre resultat. Även en alltför hög mutationssannolikhet hade sämre effektivitet vilket testet med 10% visade. I testerna av turneringsstorlek visade det sig att en turneringsstorlek på 2 gav mycket sämre resultat än de övriga storlekarna. En så hög turneringssstorlek som möjlig visade sig ge den bästa effektiviteten i simuleringen. Testen av populationsstorlek visade att små populationsstorlekar var effektivare än de större.

Kapitel 6

Avslutning

6.1 Felkällor

Vi har under arbetet med undersökningen ansträngt oss för att minimera möjliga felkällor på vårt resultat. Att helt bli av med alla möjliga felkällor är däremot inte realistiskt.

Då genetiska algoritmer till en mycket stor del bygger på slump så är det viktigt att ha en bra slumpgenerator. Vi har använt oss av den beprövade slumpgeneratorn *mt19937* från c++ biblioteket *Boost*. Den ska vara betydligt bättre på att generera slumptal likformigt fördelade över det intervall man specificerat än den generator som ingår i c++ standardbibliotek. Trots detta kan vi inte utesluta att även *mt19937* har någon avvikelse i fördelningen av slumptal som skulle kunna påverka vårt resultat.

Då algoritmen till en så stor del bygger på slump så är det dessutom viktigt att man har tillräckligt många körningar som stickprov för att kunna beräkna ett så bra medelvärde över hur stor effekt en ändring har på en viss parameter. För att minimera mätosäkerheten och statistiska avvikelser har vi därför kört evalueringar av de olika parametrarna i över 100 timmar. Detta har givit oss i snitt över 500 stickprov per test vi kört. Med 500 stickprov och en standardavvikelse på 10 får man till exempel ett standardfel på cirka 0.45. Det betyder att vi, för det flesta testerna, med en konfidensgrad på 95% kan påstå att det riktiga medelvärdet för hur många generationer som krävs för att hitta en lösning ligger inom ett intervall på plus/minus en generation från vårt beräknade medelvärde.

Det är också möjligt att de ändringar vi gjort på parametrarna mellan många av testerna har varit för små för att det ska vara någon egentlig skillnad mellan de olika körningarna. Detta speciellt för körningarna med mutationssannolikheten där det är svårt att avgöra om det är någon skillnad alls mellan många av körningarna. Även det motsatta kan vara ett problem, att vi har gjort för stora variationer på parametrarna vilket gjort att vi inte kunnat ringa in det optimala värdet på parametern vi testat med en användbar noggranhet.

Vi kan inte heller garantera att vår implementation av den genetiska algoritmen

är helt felfri, vilket skulle kunna ge märkbar påverkan på resultatet. Det är dock inte troligt att det finns några allvarligare brister i algoritmen då den löser vårt problem inom en överskådlig tid i de testfall vi kört. Det kan dock vara så att vår evalueringsbana är lite för enkel att ta sig igenom, vilket gör att antalet möjliga lösningar är tämligen stort. Detta gör att det blir svårare att evaluera exakt hur effektiv den genetiska algoritmen är.

6.2 Slutsats

Användningen av genetiska algoritmer för att lösa vårt problem fungerade över förväntan. Algoritmen lyckades hitta en lösning på omkring 10 generationer, beroende på parametrarna. Möjligen på grund av att vårt problem var för lätt att hitta en lösning till på grund av en allt för stor lösningsmängd. Genom att kombinera de optimala värdena på parametrarna borde man kunna få ner antalet generationer, innan en lösning är funnen, ytterligare. Vi kan konstatera att valet av värden på de flesta parametrarna spelar en stor roll för hur snabbt algoritmen konvergerar mot en lösning.

Slutsatsen blir att störst påverkan har selektionstrycket, då högt selektionstryck ger en mycket snabbt konvergerade genetisk algoritm. Detta fungerar bra på vårt problem, men i det allmänna fallet är det inte alltid så bra. Populationsstorleken har en stor inverkan på algoritmens effektivitet. Med en för stor populationsstorlek blir algoritmen ineffektiv räknat i mängden evalueringar som behöver göras innan en lösning blir funnen. Effektivast var en population med cirka 20 individer. För mutationssannolikheten är det svårare att avgöra ett definitivt optimalt värde då resultatet var mycket jämt för mutationssannolikheter mellan 2% och 6%. Troligen ligger det optimala värdet för vårt problem runt 3% men detta är inte helt säkerställt utifrån resultatet. Rekombinationsmetoden var det som hade minst inverkan på algoritmens effektivitet då testerna visade att det i stort sätt inte var någon reell skillnad mellan de två testade metoderna.

Avslutningsvis så kan man säga att genetiska algoritmer är en intressant metod att lösa komplexa problem med, även om den inte alltid är den optimala metoden att använda.

Litteraturförteckning

[1] Erin Catto, *Box2d, Open source 2D physics engine for games*, http://www.box2d.org/, Hämtad: 2011-04-13

[2] Randy L. Haupt, Sue Ellen Haupt, 1998, *Practical genetic algoritms*, Wiley-Interscience, 2nd ed., ISBN 0-471-45565-2

[3] Joerg Heitkoetter, David Beasley, 1995, *The Hitch-Hiker's Guide to Evolutionary Computation*, http://www.cs.cmu.edu/Groups/AI/html/faqs/ai/genetic/top.html, Publicerad: 1995-03-20, Hämtad: 2011-02-08

[4] John Markoff, 1990, *What's the Best Answer? It's Survival of the Fittest*, New York Times, http://www.nytimes.com/1990/08/29/business/business-technology-what-s-the-best-answer-it-s-survival-of-the-fittest.html, Publicerad: 1990-08-29, Hämtad: 2011-02-08

[5] Jens Maurer, Steven Watanabe, 2000, *Boost.Random, C++ library*, http://www.boost.org/doc/libs/1_46_1/doc/html/boost_random.html Hämtad: 2011-04-13

[6] Brad L. Miller , David E. Goldberg, 1995, *Genetic Algorithms, Tournament Selection and the effects of noise*, http://citeseerx.ist.psu.edu/viewdoc/download;jsessionid=621DB995CF9017353A57518149E3CAA4?doi=10.1.1.30.6625&rep=rep1&type=pdf Publicerad: 1995-07-12, Hämtad: 2011-02-08

[7] Darrell Whitley, 1993, *A Genetic Algorithm Tutorial*, Statistics and Computing, Nr 2/Volym 4, http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.38.4806&rep=rep1&type=pdf Publicerad: 1993-03-10, Hämtad: 2011-02-08

[8] Wikimedia Foundation, Inc , 2011, *Genetic algorithm*, http://en.wikipedia.org/wiki/Genetic_algorithm, Hämtad: 2011-02-08

[9] Wikimedia Foundation, Inc , 2011, *Crossover - Genetic algorithms*, http://en.wikipedia.org/wiki/Crossover_(genetic_algorithm), Hämtad: 2011-02-08

[10] Wikimedia Foundation, Inc , 2011, *Evolution*,
`http://en.wikipedia.org/wiki/Evolution`, Hämtad: 2011-03-29

[11] Wikimedia Foundation, Inc , 2011, *Hill Climbing*,
`http://en.wikipedia.org/wiki/Hill_climbing`, Hämtad: 2011-04-03