πGE

Grammaticale evolutie met behulp van sGE en πGE

Joris Steenstra, 3252477

June 24, 2015

Dr. G.A.W. Vreeswijk
7,5 ECTS

Contents

1 Inleiding

Binnen de kunstmatige intelligentie wordt voortdurend gezocht naar nieuwe methoden om computers zodanig intelligent te maken zodat ze zelf oplossingen kunnen zoeken voor een probleem. In de jaren '50 werd er inspiratie gehaald uit de natuur, waardoor genetisch programmeren ontstond, een techniek waar met behulp van kunstmatige evolutie getracht werd programma's zichzelf te laten verbeteren. Eind jaren '90 ontstond er een nieuwe wijze van genetisch programmeren, grammaticale evolutie, waarbij via een kunstmatige grammatica keuzes worden gemaakt om een oplossing te vinden. Aanleiding voor deze scriptie is het artikel van O'Neill, (O'Neill et al., 2004). In dit artikel wordt een variant van de standaard grammaticale evolutie (sGE) aangeboden: een positie onafhankelijke grammaticale evolutie (*position independent grammatical evolution*, (πGE)). Volgens de schrijvers van dit artikel is deze variant een doorbraak in het veld van de grammaticale evolutie omdat het een grotere flexibiliteit biedt dan sGE. De auteurs onderbouwen hun artikel door zowel sGE als πGE toe te passen op een aantal experimenten waarbij πGE beduidend beter uit de tests komt.

Deze scriptie gaat in op de verschillen tussen πGE en sGE, met de bedoeling te onderzoeken of πGE daadwerkelijk efficiënter is. De indeling van de scriptie is als volgt: Allereerst bekijk ik in hoofdstuk 2 de theorie, waarbij ik in 2.1 eerst grammaticale evolutie in het algemeen uiteenzet. Vervolgens laat ik in hoofdstuk 2.2 het verschil tussen πGE en sGE zien. Hierbij wordt met name ingegaan op de achterliggende werkwijzen. Naast het theoretische gedeelte is voor deze scriptie gekozen om in Java twee testproblemen op te lossen met zowel πGE als sGE. De code hiervan is te vinden op GitHub (Steenstra, 2015). De twee experimenten zijn Mastermind en de Benadering van een formule, deze problemen worden vaker als benchmark gebruikt binnen de kunstmatige intelligentie. In hoofdstuk 3 worden de gebruikte methoden uiteengezet, waarna de resultaten in hoofdstuk 4 besproken worden. Uiteindelijk worden in het vijfde hoofdstuk (de conclusie) de resultaten met elkaar vergeleken, alsmede met die van het artikel van O'Neill, om vervolgens antwoord te geven op de hoofdvraag: Hoe werkt πGE en in welke situaties presteert πGE beter dan sGE?

2 Theorie

2.1 Grammaticale Evolutie

Grammaticale Evolutie (GE) is een methode om computerprogramma's te laten evolueren, onafhankelijk van de taal waarin ze geschreven zijn. In plaats van het presenteren van de programma's in een boomstructuur, zoals in genetisch programmeren wordt gedaan, wordt bij GE het genoom in een lineaire reeks van tekens gerepresenteerd (O'Neill and Ryan, 2003). De erfelijke informatie wordt gebruikt om met behulp van een grammatica een programma te vormen in een *genotype-phenotype mapping proces*. Ieder individu bezit een genoom, bestaande uit een reeks binaire getallen. Deze reeks wordt opgedeeld in een aantal codons met een vaste lengte, waardoor de bits die samen een codon vormen, samen staan voor een waarde. Eveneens wordt er gebruik gemaakt van een *Backus Naur Form* (BNF), dit is een reeks van herschrijfregels, (Backus et al., 1960). Iedere regel heeft een kop en een staart; de kop is de invoer en de staart bestaat uit alle mogelijke uitvoeren, gescheiden door een 'of' symbool. Iedere kop is een *Non-Terminal*, te herkennen aan de <haakjes>. Zoals de term al suggereert, zijn de Non-Terminals nog niet af, ze dienen uiteindelijk vervangen te worden door *Terminals*. Een vaste regel van de BNF is dat deze altijd begint met een startsymbool, de Non-Terminal <s>. Hieronder een BNF met drie herschrijfregels, waarmee alle natuurlijke getallen gevormd kunnen worden. Ook is het met deze grammatica mogelijk getallen te vormen die beginnen met een aantal nullen. In de praktijk zou er voor gekozen kunnen worden om deze te negeren en bijvoorbeeld 007 als het getal 7 te interpreteren.

```
<s>     ::= <digit> | <digit><s>
<digit> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
```

In dit voorbeeld wordt het getal 3 gevormd door het startsymbool <s> te vervangen door <num>, vervolgens <num> door <digit> en <digit> door 3. Het getal 127 kan gevormd worden door

```
<s> → <num> → <digit><num> → <digit><digit><num> → <digit><digit><digit> → 127
```

Herschrijfregels zijn dus een manier om een Non-Terminal te vervangen door iets anders: ofwel een Non-Terminal ofwel een Terminal. Om een keuze te maken binnen een herschrijfregel in GE, wordt de codonwaarde gelezen, geïnterpreteerd en in de volgende formule geplaatst, waarbij c staat voor de codonwaarde en k voor het aantal keuzemogelijkheden:

$$c \pmod{k}$$

In het vervolg zal een andere notatie gebruikt worden, zoals die ook in de meeste programmeertalen gebruikt wordt:

```
Codonwaarde % aantal keuzemogelijkheden in deze herschrijfregel
```

In deze formule staat % voor modulus; het restgetal bij deling. In bovenstaande BNF voor natuurlijke getallen zien we dat de <num> herschrijfregel twee keuzemogelijkheden heeft, en de <digit> herschrijfregel tien. Wanneer we de (willekeurig gekozen) codonwaarde 37 in de formule voeren, volgt de volgende berekening:

```
Regel:                      <digit> ::= 0|1|2|3|4|5|6|7|8|9
Formule:                    codonwaarde % aantal keuzemogelijkheden
Codonwaarde:                37
Aantal keuzemogelijkheden: 2
Berekening:                 37 % 2 = 1
```

Doordat de oplossing één is, wordt de tweede regel gekozen. De regelnummering begint namelijk bij nul, aangezien het restgetal bij deling ook nul kan zijn. In het volgende voorbeeld zal dan ook voor de achtste keuzemogelijkheid worden gekozen.

```
Formule:                    codonwaarde % aantal keuzemogelijkheden
Codonwaarde:                37
Aantal keuzemogelijkheden: 10
Berekening:                 37 % 10 = 7
```

Het is duidelijk dat de waarden die een codon moet kunnen aannemen, groter moet zijn dan het maximale aantal keuzemogelijkheden binnen de BNF (per regel), aangezien anders niet alle mogelijkheden gekozen zouden kunnen worden. Wanneer het aantal keuzes bijvoorbeeld drie zou zijn, zou één bit dus niet volstaan, maar ook twee bits zouden problemen op kunnen leveren:

```
<KEUZE> ::=
            00 → A
            01 → B
            10 → C
            11 → ?
```

Een voor de hand liggende oplossing is nu te kiezen voor een A, B, of C op de plek van het vraagteken. Het nadeel hiervan is dat er een duidelijke voorkeur wordt gegeven aan een bepaalde keuze. In dit geval zou bijvoorbeeld A 50% kans hebben, en B en C slechts 25%. Dit kan handig zijn in sommige gevallen, maar kan het programma een richting in sturen die wellicht niet de bedoeling is van de programmeur. Een andere oplossing zou binnen de BNF plaats kunnen vinden; door de '?' in het voorbeeld te vervangen door <KEUZE>, waardoor het volgende codon de keuze zou bepalen. Het nadeel hiervan is dat het de mogelijkheid opent tot een eindeloze lus.

Het feit dat meerdere codons voor een enkele functie kunnen coderen zorgt eveneens voor problemen, aangezien het aantal codons dan variabel is, en vanwege het veranderen van de context een andere betekenis kunnen krijgen.

Verder zou het tot gevolg kunnen hebben dat een latere mutatie extra nadeel ondervindt van de codons die terugverwijzen naar hetzelfde gedeelte, bijvoorbeeld:

```
.. 11 11 11 10 ..
    ↓
.. 10 11 11 10 ..
```

Na de mutatie zullen alle codons die hierna komen voor een andere functie coderen dan voorheen, waardoor het in feite de randomisering van de gehele string tot gevolg heeft, hetgeen hoogstwaarschijnlijk zal leiden tot een lagere fitness en daarmee verwijdering uit de *gene-pool*. In de biologie worden dit soort mutaties *frameshift-mutations* genoemd.

Als oplossing voor het probleem stel ik voor een groot genoeg getal te kiezen. Het aantal waarden dat een n-bit codon kan hebben is 2^n. Deze waarde dient uiteraard groter te zijn dan het aantal keuzes K dat een regel heeft. De waarde van K kan in sommige gevallen redelijk groot zijn; een herschrijfregel voor het alfabet zou bijvoorbeeld al 26 keuzes hebben. $2^n \% K =$ het aantal keuzes met een grotere kans dan de rest. Het maximale verschil tussen de keuzes is $1/(2^n \div K)$, waarbij $\div$ staat voor het quotiënt.

$$1/(2^2 \div 3) = 1$$
$$1/(2^4 \div 3) = 0.2$$
$$1/(2^8 \div 3) = 0.012$$

Zelfs in het geval van het alfabet zal dit slechts leiden tot een kansverschil van 1/9 tussen de verschillende keuzes. Alhoewel dit geen verwaarloosbaar verschil is, wegen de voordelen van een groter getal niet op tegen de toenemende geheugenkosten. Er is in dit onderzoek dus gekozen om een codongrootte van een byte aan te houden. Hierbij kunnen de codons dus 256 verschillende waarden aannemen.

```
Binaire code:   00000001 00000000 10000001 11111111
Codonwaarde:       1        0       129       255
```

Een nadeel van GE is dat er een sterke relatie bestaat tussen de volgorde van de codons in het genotype en het resulterende fenotype, (Fagan et al., 2010). Wanneer een enkele bit verandert, kan het totale genotype veranderen, zeker wanneer deze bit aan het begin van het genoom staat. Dit is het beste te demonstreren aan de hand van een voorbeeld. Neem de volgende BNF, die codeert voor een reeks a's en b's.

```
<s>       ::=  <s><letter>|<letter>
<letter>  ::=  a|b
```

Wanneer het genotype 00010000 (codongrootte 1) is, is het resulterende genotype:

```
<s>
0 <s><letter>
0 <s><letter><letter>
0 <s><letter><letter><letter>
1 <letter><letter><letter><letter>
0 a<letter><letter><letter>
1 ab<letter><letter>
0 aba<letter>
1 abab
```

Wanneer er slechts een enkele bit in het genotype verandert, 10010000, is het resulterende genotype:

```
<s>
1 <letter>
0 a
0
1
0
1
0
1
```

Wanneer alle bits in het genotype dus een even grote kans op mutatie hebben, is de kans groot dat een enkele mutatie het totale individu verandert, vooral wanneer het een mutatie is in het eerste stuk van het genoom. Het is te vergelijken met een routebeschrijving; wanneer je de eerste afslag rechtsaf gaat in plaats van links, is de eindbestemming heel anders, ook al volg je de rest van de aanwijzingen precies.

Dit is het grootste nadeel van 'normale' GE algoritmen. De betekenis van een codon hangt niet alleen af van de waarde die het heeft, maar evenveel van de positie binnen het genoom en de waarde van alle voorgaande codons. Het veranderen van het n-de codon, plaatst alle codons met $i > n$ mogelijk in een andere context, waardoor dezelfde waarde een totaal andere betekenis kan krijgen. Een oplossing hiervoor is het maken van een aanpassing aan het mapping-proces. De 'normale' gang van zaken is het verwerken van het genoom van links naar rechts; bij iedere stap wordt de meest linkse non-terminal vervangen aan de hand van een herschrijfregel. Er zijn een aantal variaties op het standaard mapping-proces bedacht die trachten dit op te lossen, zoals Chorus (Ryan, Azad, et al., 2002), GAuGE (Ryan, Nicolau, and O'Neill, 2002) en Position Independent Gramatical Evolution (πGE) (O'Neill et al., 2004), waarbij mijns inziens πGE de beste oplossing geeft.

2.2 Positie-onafhankelijke grammaticale evolutie

In een standaard GE *genotype-fenotype mapping process* wordt de meest linkse non-terminal altijd als eerste uitgebreid. Het *mapping proces* van *Position Independent Grammatical Evolution* (πGE) verschilt van GE doordat de mogelijkheid om de volgorde van non-terminals vast te stellen en te wijzigen is uitgebreid. Een πGE codon bestaat uit twee waarden, *nont* (non-terminal index) en *rule* (regel-index), waarbij nont en rule worden gerepresenteerd door N bits (in dit geval acht bits). Het eerste gedeelte (nont) bepaalt welke non-terminal vervangen moet worden, in tegenstelling tot standaard GE, waar altijd van links naar rechts gewerkt wordt. Het tweede gedeelte (rule) bepaalt waar de non-terminal door vervangen moet worden. Het genoom bestaat in πGE uit een vector van deze paren en zou er dus als volgt uit kunnen zien:

```
Codonwaarde:       222      131      145      170
Binaire code:    11011110 10000011 10010001 10101010
Binaire code:    00000010 00000000 10000001 11111111
Codonwaarde:       2        0        129      255
```

In πGE wordt de status van het proces bij iedere stap geanalyseerd door het aantal aanwezige non-terminals te tellen. Wanneer er meer dan één non-terminal aanwezig is, worden alle aanwezige non-terminals geteld en door middel van de volgende formule wordt bekeken welke non-terminal uitgebreid moet worden:

$$c \pmod{n}$$

Waarbij c staat voor de codonwaarde en n voor het aantal Non-Terminals. Ofwel:

```
NT = Codonwaarde van het nont-codon % aantal Non-Terminals
```

Wanneer middels deze formule is bepaald welke non-terminal uitgebreid gaat worden, wordt dezelfde formule als bij GE toegepast op de non-terminal. πGE verschilt dus van GE door het feit dat niet altijd de meest linkse non-terminal wordt vervangen. In plaats daarvan wordt de keuze gemaakt aan de hand van het genoom.

Om de werkwijze van πGE toe te lichten, volgt een voorbeeld met een gekozen BNF die codeert voor een logische formule. In deze BNF zitten drie non-terminals, die ieder hun eigen keuzemogelijkheden hebben. Wanneer naar de non-terminal <s> wordt gekeken en het restgetal dat uit de formule komt is 0, dan wordt <s> vervangen door <logical>. Is het getal 1, dan wordt <s> vervangen door <boolean>

```
BNF:
<s>          ::=
    0        <logical>
    1      | <boolean>

<logical>    ::=
    0        not(<boolean>)
    1      | and(<boolean>, <boolean>)
    2      | or(<boolean>, <boolean>)

<boolean>    ::=
    0        TRUE
    1      | FALSE
    2      | <logical>
```

Naast deze BNF hebben we het volgende genoom:

```
11001000010111000001000010100000111111001011000111011100000000010100101010100010011
10011011000001001101111111101010
```

Deze string wordt opgedeeld in reeksen van lengte acht (codons) en opgedeeld in paren. Tevens wordt het cijferkundige equivalent uitgerekend. Vervolgens kunnen we beginnen met het *mapping proces* van genotype naar fenotype.

```
 NONT      RULE
(11001000, 01011100)    (200,  92)
(00010000, 10100000)    ( 16, 160)
(11111100, 10110001)    (252, 177)
(11011100, 00000010)    (220,   2)
(10010101, 01000100)    (149,  68)
(11001101, 10000010)    (205, 130)
(01101111, 11101010)    (111, 234)
```

Stap 0: <s> is het startpunt van het *mapping process*.

```
<s>
```

Stap 1: Het eerste codon is (11001000, 01011100) (200, 92). De 200 wordt niet gebruikt aangezien <s> de enige non-terminal is, waardoor er niet gekozen hoeft te worden. Om te kijken waar de <s> door vervangen moet worden, wordt het tweede getal gebruikt en het restgetal berekend: 92%2 = 0, dus <s> wordt vervangen door <logical>

```
<logical>
```

Stap 2: Het tweede codon is (00010000, 10100000) (16, 160). De 16 wordt niet gebruikt want <logical> is de enige non-terminal. Het aantal keuzes voor <logical> is 3. De berekening wordt dan dus 160%3 = 1. <logical> wordt vervangen door and(<boolean>, <boolean>).

```
and(<boolean>, <boolean>)
```

Stap 3: Het derde codon is (11111100, 10110001) (252, 177). and(<boolean>, <boolean>) bevat twee non-terminals dus aan de hand van de eerste waarde van het volgende codon wordt bepaald welke uitgebreid wordt. De berekening wordt dus 252%2 = 0 waardoor de eerste non-terminal vervangen zal worden. De eerste non-terminal <boolean> heeft drie keuzes, dus 177%3 = 0. <boolean> wordt TRUE en and(<boolean>, <boolean>) wordt dus and(TRUE, <boolean>).

```
and(TRUE, <boolean>)
```

Stap 4: Het vierde codon is (11011100, 00000010) (220, 2). and(TRUE, <boolean>) bevat één non-terminal. De non-terminal <boolean> heeft drie keuzes waardoor uit de berekening (2%3 = 2) komt dat <boolean> vervangen wordt door <logical>.

```
and(TRUE, <logical>)
```

Stap 5: Het vijfde codon is (10010101, 01000100) (149, 68). and(TRUE, <logical>) bevat één non-terminal en <logical> heeft drie keuzes. 68%3 = 2, dus <logical> wordt or(<boolean>, <boolean>).

```
and(TRUE, or(<boolean>, <boolean>))
```

Stap 6: Het zesde codon is (11001101, 10000010) (205, 130). and(TRUE, or(<boolean>, <boolean>)) bevat twee non-terminals. Aan de hand van de eerste waarde van het volgende codon wordt bepaald welke non-terminal gekozen wordt. De berekening wordt dus 205%2 = 1. De tweede non-terminal wordt dus vervangen. <boolean> heeft drie keuzes, dus 130%3 = 1. Dus <boolean> wordt vervangen door FALSE.

```
and(TRUE, or(<boolean>, FALSE))
```

Stap 7: Het zevende codon is (01101111, 11101010) (111, 234). and(TRUE, or(<boolean>, FALSE)) bevat één non-terminal. <boolean> heeft drie keuzes. 234%3 = 0, dus <boolean> wordt vervangen door TRUE.

```
and(TRUE, or(TRUE, FALSE))
```

Stap 8: and(TRUE, or(TRUE, FALSE)) bevat geen non-terminals. Hiermee is de conversie van genotype naar fenotype afgerond.

```
Genotype: 1100100001011100000100001010000011111100101100011101110000000010100101010
          1010001001100110110000001001101111111101010
Fenotype: and(TRUE, or(TRUE, FALSE))
```

Het proces eindigt nu doordat er geen non-terminals meer over zijn; er is niets meer te vervangen, dus de rest van het genoom wordt niet meer gebruikt. Merk op dat er een groot aantal verschillende genotypen tot hetzelfde fenotype zouden kunnen leiden. In stap 1 werd het eerste getal niet gebruikt, omdat er maar één keuze was. Dit getal had dus net zo goed een ander getal kunnen zijn. Ook het tweede codon had een aantal andere waarden aan kunnen nemen, namelijk alle even getallen, aangezien die allemaal 0 als uitkomst geven wanneer ze modulus 2 gedaan worden. Deze keuzemogelijkheden zorgen ervoor dat er ruimte is voor *genetische drift*, een bekend fenomeen in de evolutionaire biologie, waarbij delen van het DNA van een soort veranderen die geen directe weerslag hebben op de individuen.

3 Methode

Om er achter te komen of πGE efficiënter is dan sGE, worden er twee experimenten toegepast op sGE en πGE. Hierbij is het van belang een eerlijke vergelijking te maken tussen deze verschillende methoden, waarbij alle variabelen zoveel mogelijk hetzelfde dienen te blijven. Eén van deze variabelen is de lengte van het genoom. Een probleem hierbij is echter dat de verschillende methoden op een andere wijze gebruik maken van de variabelen in het genoom. Waar sGE gebruik maakt van een enkele codon per uitbreidings-stap, gebruikt πGE er twee: zowel de keuze-codons die sGE ook gebruikt, als positie-codons die bepalen welke non-terminal er uitgebreid gaat worden.

Er zou voor gekozen kunnen worden om de lengte van het genoom gelijk te houden. Het feit dat πGE meer codons gebruikt, is de prijs die het betaalt om (mogelijk) efficiënter te zijn. Toch is ook dat niet helemaal eerlijk, aangezien in dat geval een RUN voor sGE veel langer duurt dan een voor πGE, waardoor in feite sGE meer rekentijd krijgt per stap.

Het voor de hand liggende alternatief is het genoom van πGE een dubbele lengte toe te kennen, maar dit zou een oneerlijk voordeel kunnen zijn voor πGE, hoewel dit voordeel een stuk kleiner zou zijn dan andersom, aangezien er slechts een enkele berekening extra gebruikt wordt per stap. Omdat deze twee keuzes beide voordelen voor één van de twee methodes met zich meebrengen, is er gekozen om beide mogelijkheden uit te werken, zodat een duidelijk beeld wordt verkregen van de voordelen. Er is gekozen om twee benchmarkproblemen uit te werken op twee manieren: met dezelfde genoomlengte voor zowel sGE als πGE (methode 1) en met een dubbele genoomlengte voor πGE ten opzichte van sGE (methode 2).

Hier volgt pseudocode om het verschil te laten zien tussen de versie van πGE en sGE die is gebruikt in de experimenten:

Standaard GE

```
for(codon_index ci; ci += 1)
{
  if(aantal_non_terminals == 0)
  {
    return individu
  }

  // ci = codon index

  int CHOICE = codonArray[ci]
  nont_index = 0 // altijd de eerste

  int AO = get_outputs(nont_index)
  // selecteer de regel die hoort
  // bij de gekozen non_terminal
  // en kijk hoeveel mogelijke
  // outputs deze regel heeft (AO)

  keuze_index = CHOICE % AO
  herschrijf(nont_index, keuze_index)
}
```

πGE

```
for(codon_index ci; ci += 2)
{
  if(aantal_non_terminals == 0)
  {
    return individu
  }

  // ci = codon index
  int RULE = codonArray[ci]
  int CHOICE = codonArray[ci+1]
  nont_index = RULE % aantal_NONTs

  int AO = get_outputs(nont_index)
  // selecteer de regel die hoort
  // bij de gekozen non_terminal
  // en kijk hoeveel mogelijke
  // outputs deze regel heeft

  keuze_index = CHOICE % AO
  herschrijf(nont_index, keuze_index)
}
```

Merk op dat de code nagenoeg gelijk is, het enige verschil zit in het gebruik van twee variabelen in plaats van een variabele en een constante.

3.1 Mastermind

Het originele Mastermind, dat dateert uit 1972, wordt gespeeld met zes kleuren op vier posities. Deze variant kent dus $6^4 = 1296$ mogelijkheden. In 1975 is een moeilijkere variant bedacht, waarbij er wordt gespeeld met acht kleuren en vijf posities, hetgeen zorgt voor $8^5 = 32768$ mogelijke combinaties; deze variant zal ik gebruiken.

Het programma genereert vijf willekeurige cijfers (kleuren) uit een pool van acht, waarna een individu een score krijgt aan de hand van de correctheid van de cijfers. Drie punten voor het juiste cijfer op de juiste plek en één punt voor het juiste cijfer op de verkeerde plek. Er zijn dus maximaal $5 * 3 = 15$ punten te behalen voor de juiste combinatie, maar uiteindelijk wordt de score door 15 gedeeld zodat een maximale fitness van één te behalen is. Het experiment is uitgevoerd met een populatie van 30 individuen die gedurende (maximaal) 30 generaties evolueren en stoppen zodra ze de maximale fitness van vijftien punten behaald hebben. Verder is het experiment 100 keer herhaald, waarna de gemiddelde fitness per generatie is uitgerekend.

In onderzoek van O'Neill wordt op verschillende manieren het experiment gemanipuleerd in het voordeel van πGE. Zo wordt in het voorbeeld van Mastermind de volgende methode toegepast:

> In this problem the code breaker attempts to guess the correct combination of coloured pins in a solution. When an evolved solution to this problem (i.e. a combination of pins) is to be evaluated, it receives one point for each pin that has the correct colour, regardless of its position. If all pins are in the correct order than [sic] an additional point is awarded to that solution. This means that ordering information is only presented when the correct order has been found for the whole string of pins. A solution, therefore, is in a local optimum if it has all the correct colours, but in the wrong positions, (O'Neill et al., 2004).

Hier wordt dus pas een extra punt aan de score toegekend wanneer alle pinnetjes op de juiste plek staan. Dit is niet conform de regels van Mastermind, waarbij met behulp van witte en zwarte/rode pinnetjes direct al informatie wordt gegeven over de correctheid van de positie van de pinnetjes. Dat dit in het voordeel werkt van πGE is duidelijk; deze methode is namelijk sterk in het herordenen van het eindresultaat, terwijl sGE de code in feite in één keer goed moet raden aangezien er op deze manier geen relatie is tussen (bijvoorbeeld) 01230 en 23001.

Ook laat de door O'Neill *et al.* gebruikte BNF illegale pogingen toe; met meer of minder dan het gewenste aantal pinnetjes. Dit is volgens de officiële spelregels niet toegestaan.

```
<pin> ::= <pin> <pin> | 0 | 1 | 2 | 3
```

Ik heb er daarom voor gekozen om zelf een andere BNF te schrijven, die enkel legale pogingen toe laat. Ook geeft de fitnessfunctie een score die informatie geeft over zowel de correctheid van de positie als de kleur van de pinnetjes.

```
<s> ::= <c>_<c>_<c>_<c>_<c>
<c> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7
```

Deze grammatica zorgt er voor dat er altijd vijf pinnetjes gegenereerd worden, zodat in ieder geval legale pogingen worden gedaan. Ook laat het ruimte toe voor meerdere kleuren zodat de moeilijkere variant van Mastermind, met acht kleuren, gespeeld kan worden.

3.2 Benadering van een formule

In het tweede experiment is met behulp van een BNF eerst een willekeurig individu gegeneerd in de vorm van een formule: $((x + (((x/x) * ((x + x) * x)) - x))/\cos(\sin(-4)))$. In deze formule worden een aantal waarden gevoerd, die vervolgens eveneens een aantal waarden als uitvoer hebben. Deze waarden worden opgeslagen. Vervolgens worden er aan de hand van deze BNF individuen gegenereerd die ook de vorm van een formule hebben. Deze krijgen dezelfde waarden ingevoerd en hun uitvoer wordt vergeleken met die van de originele formule. Weinig verschil is een hoge fitness.

De gebruikte BNF is de volgende:

```
<expr>  ::= <func> | <val> | x | (<expr> <op> <expr>)
<op>    ::= + | - | * | /
<func>  ::= sin(<expr>) | cos(<expr>) | tan(<expr>) | sqrt(<expr>) | (x <op> x)
<val>   ::= x | <num> | -<num>
<num>   ::= <digit> | <digit><num>
<digit> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
```

De resulterende programma's zijn dus van de vorm: $ax^2 + bx + c$. De fitnessfunctie berekent voor deze formule voor ieder punt het (absolute) verschil tussen de doelwaarde en de waarde die door het programma wordt gegeven. De som van deze verschillen wordt van de maximum-fitness afgetrokken (zodat een hoge waarde een hoge fitness weergeeft).

3.3 Methode van visualisatie

De experimenten produceerden een grote hoeveelheid rauwe data, in de vorm van genotypen met de bijbehorende fitness. Ieder genotype had een nummer wat aangaf tot welke run hij behoorde en tot welke generatie. Vervolgens werd van iedere generatie diegene met de hoogste fitness gekozen en werd over alle runs het gemiddelde van deze getallen per generatie berekend. Bij Mastermind werden 100 runs gedraaid met ieder 30 generaties en een populatie van 30 individuen. Ieder punt op de grafiek is berekend met de volgende formule:

```
Run 1,   Generatie 1: kies individu met hoogste fitness
Run 2,   Generatie 1: kies individu met hoogste fitness
...
Run 100, Generatie 1: kies individu met hoogste fitness
```

Deze getallen zijn bij elkaar opgeteld en door het aantal runs gedeeld (100). Het resulterende getal is het eerste punt op de grafiek. Het tweede punt op op de grafiek werd op dezelfde wijze berekend, alleen werd hier uit individuen van de tweede generatie gekozen, en zo verder tot het laatste punt, generatie 30. Bij experiment 2, de benadering van een formule, werd dezelfde methode toegepast, alleen werden hier 500 runs gedraaid, met als doel de grafiek te vereffenen.

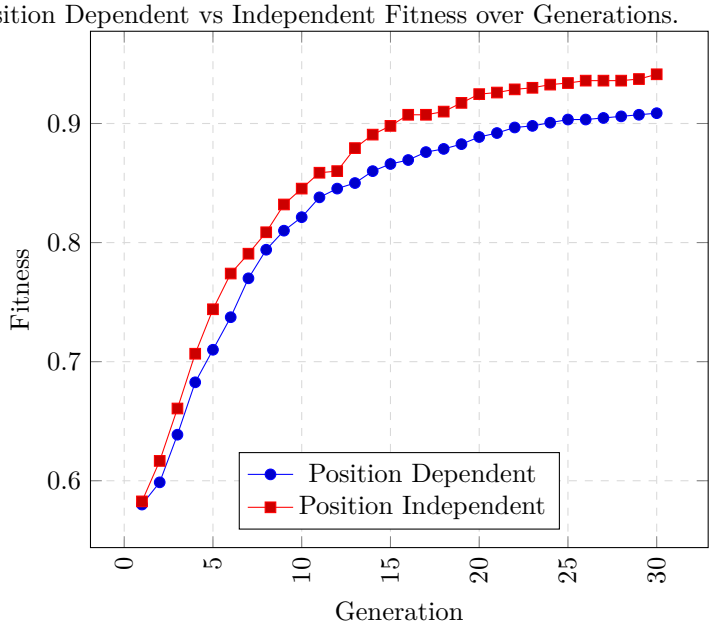
Beide experimenten zijn herhaald met een dubbele codongrootte voor πGE, om te compenseren voor de wijze van berekenen, zoals op de vorige pagina besproken werd. Dit resulteert in twee grafieken per experiment.

4 Resultaten

4.1 Mastermind

4.1.1 Methode 1: gelijke genoomlengte

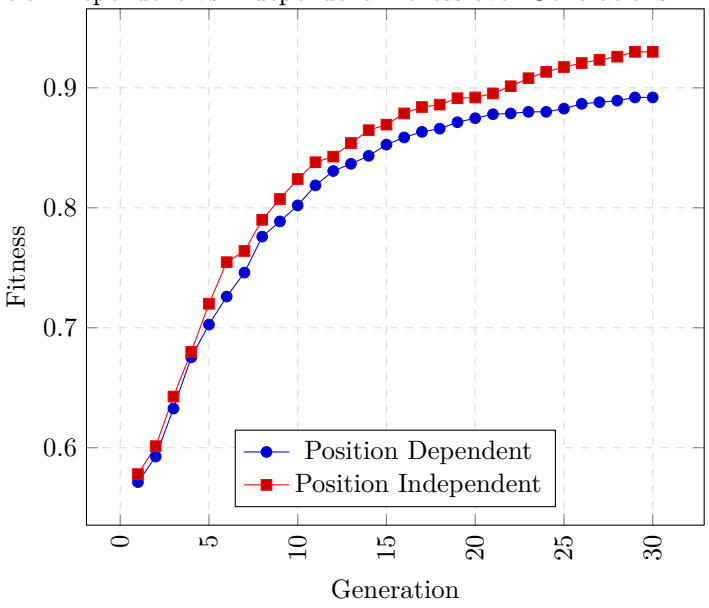
Position Dependent vs Independent Fitness over Generations.




```
SETTINGS:
population size = 30
# runs          = 100
# generations   = 30
# codons PD     = 50
# codons PI     = 50
codonsize       = 8
mutation rate   = 0.01
```

4.1.2 Methode 2: dubbele genoomlengte voor πGE

Position Dependent vs Independent Fitness over Generations.

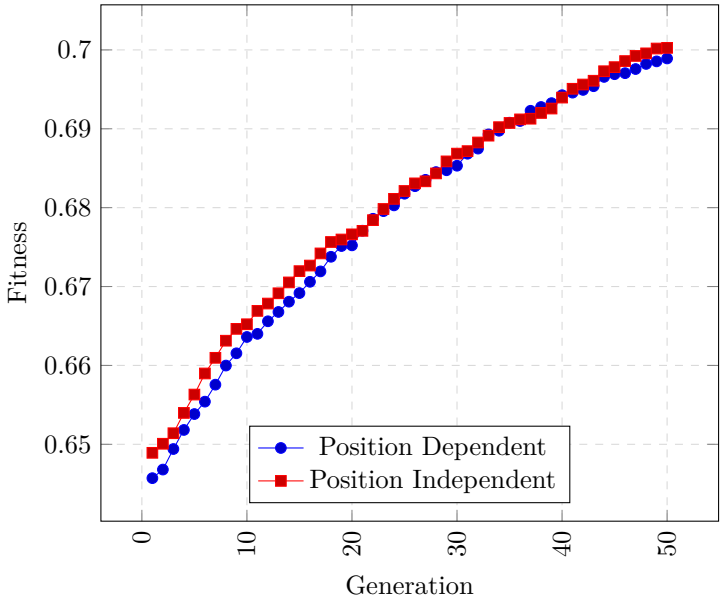



```
SETTINGS:
population size = 30
# runs          = 100
# generations   = 30
# codons PD     = 50
# codons PI     = 100
codonsize       = 8
mutation rate   = 0.01
```

Zoals in bovenstaande grafieken te zien is, presteert πGE iets beter dan sGE bij Mastermind. Het variëren van de codongrootte lijkt weinig invloed te hebben op de resultaten. Een mogelijke verklaring hiervoor is dat de hoeveelheid codons groot genoeg is om voor de juiste genomen te coderen.

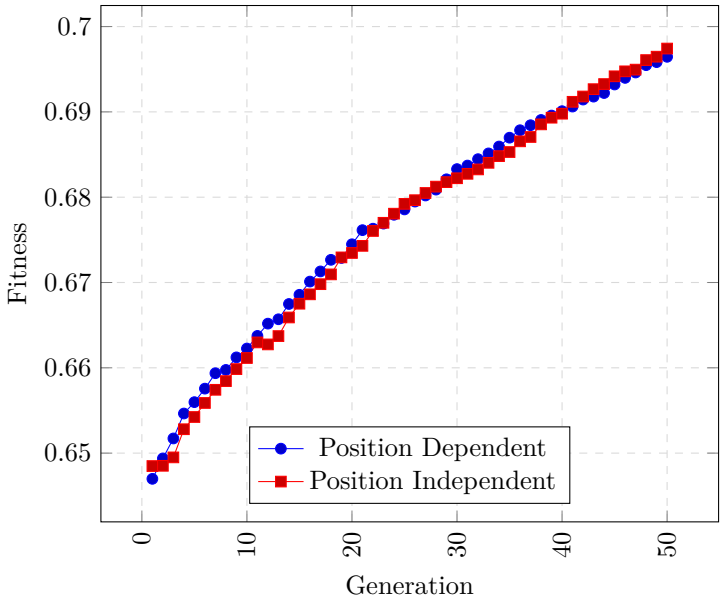
4.2 Benadering van een formule

4.2.1 Methode 1: gelijke genoomlengte




```
SETTINGS:
population size = 30
# runs          = 500
# generations   = 50
# codons PD     = 100
# codons PI     = 100
codonsize       = 8
mutation rate   = 0.01
```

4.2.2 Methode 2: dubbele genoomlengte voor πGE




```
SETTINGS:
population size = 30
# runs          = 500
# generations   = 50
# codons PD     = 50
# codons PI     = 100
codonsize       = 8
mutation rate   = 0.01
```

Uit bovenstaande grafieken blijkt, net als bij Mastermind, dat er niet veel verschil is tussen een gelijke of een dubbele genoomlengte. Wat hier echter wel duidelijk anders is dan bij Mastermind, is dat er bij dit experiment niet veel verschil is tussen πGE en sGE; ze presteren vrijwel gelijk aan elkaar. Waar bij Mastermind πGE duidelijk beter presteerde, is er bij de benadering van de gekozen formule geen duidelijk 'betere' methode aan te wijzen.

5 Conclusie

Grammaticale evolutie is een methode om met behulp van een kunstmatige grammatica (BNF) programma's te genereren en deze aan de hand van een fitness functie een score te geven, al naar gelang hun prestaties. Vervolgens worden de programma's met een hoge fitness als basis gebruikt voor een reeks nieuwe programma's, waarna dit proces zich herhaalt. Bij πGE is er een aanpassing gemaakt in de manier waarop het genoom van de programma's wordt omgezet van genotype tot fenotype met als doel minder afhankelijk te zijn van de volgorde van de waarden in het genoom. De reden voor die aanpassing is dat als er een mutatie plaats vindt in het begin van het genotype bij sGE dit alle volgende waarden in een andere context plaatst. Hierdoor hebben veranderingen aan het begin van het genotype een veel grotere impact dan veranderingen aan het eind.

In dit paper is onderzocht op welke manier πGE beter presteert dan sGE aan de hand van twee testproblemen. Na aanleiding van het onderzoek van O'Neill *et al.* is er bij het eerste experiment gekeken naar de prestaties bij het spel Mastermind. Waar O'Neill *et al.* gebruik maakten van methoden die tegen de spelregels van Mastermind in gaan en in het voordeel van πGE werken, heb ik die praktijken proberen te vermijden in de door mij uitgevoerde experimenten. Uit de resultaten van het eerste experiment bleek echter dat ook zonder deze methoden er een gelijk of beter resultaat gehaald werd door πGE. De dubbele genoomlengte had geen invloed op de resultaten, slechts op de duur van de tests. De onderzoeken in het artikel van O'Neill *et al.* zijn naar mijn mening enigszins verdraaid door het gebruik van bepaalde waarden. De indruk wordt gegeven dat het spel Mastermind getest wordt, terwijl er eigenlijk een variant getest wordt, met regels die in het voordeel van πGE werken. Zo wordt er eerst geselecteerd op het bevatten van de juiste cijfers en vervolgens pas op de volgorde.

In het tweede experiment is er een formule gegenereerd die moest worden benaderd door beide methodes. Uit de resultaten van het tweede experiment bleek dat πGE ongeveer gelijk presteert met sGE. Ook hier bleek het verdubbelen van de genoomlengte weinig invloed te hebben op het eindresultaat. Een mogelijke verklaring hiervoor is dat de genoomlengte groot genoeg was en dat een grotere genoomlengte dus geen voor- of nadelen bood bij het vinden van een oplossing.

πGE heeft desondanks betere resultaten en kan naar mijn mening een goede vervanging zijn voor de standaard methode. Zeker gezien het feit dat het een relatief simpele aanpassing is aan het algoritme. πGE zou in een groter onderzoek getest kunnen worden op diverse benchmarkproblemen binnen de AI, om op deze manier te ontdekken waar de krachten precies liggen. Eventueel vervolgonderzoek zou zich bijvoorbeeld kunnen richten op het doen van een meta-test waarbij een derde variabele (naast fitness en generatie) gevarieerd wordt. Op deze manier kan er een driedimensionale grafiek worden gegenereerd waar uit te halen zal zijn bij welke waarden er beter gekozen kan worden voor sGE en voor πGE. Mogelijke kandidaten voor deze derde variabele zijn genoomlengte, complexiteit (bijvoorbeeld bij Mastermind het aantal kleuren of posities), de kans op mutatie of de grootte van de populatie.

6 Literatuur

References

Backus, J. W. et al. (1960). "Report on the Algorithmic Language ALGOL 60". In: *Commun. ACM* 3.5. Ed. by Peter Naur, pp. 299–314. ISSN: 0001-0782. DOI: 10.1145/367236.367262. URL: http://doi.acm.org/10.1145/367236.367262.

Fagan, David et al. (2010). "An Analysis of Genotype-Phenotype Maps in Grammatical Evolution". English. In: *Genetic Programming*. Ed. by AnnaIsabel Esparcia-Alcázar et al. Vol. 6021. Lecture Notes in Computer Science. Springer Berlin Heidelberg, pp. 62–73. ISBN: 978-3-642-12147-0. DOI: 10.1007/978-3-642-12148-7_6. URL: http://dx.doi.org/10.1007/978-3-642-12148-7_6.

O'Neill, Michael and Conor Ryan (2003). *Grammatical Evolution: Evolutionary Automatic Programming in an Arbitrary Language (Genetic Programming)*. Springer. ISBN: 1402074441. URL: http://www.amazon.com/Grammatical-Evolution-Evolutionary-Automatic-Programming/dp/1402074441?SubscriptionId=0JYN1NVW651KCA56C102&tag=techkie-20&linkCode=xm2&camp=2025&creative=165953&creativeASIN=1402074441.

O'Neill, Michael et al. (2004). "piGrammatical Evolution". English. In: *Genetic and Evolutionary Computation – GECCO 2004*. Ed. by Kalyanmoy Deb. Vol. 3103. Lecture Notes in Computer Science. Springer Berlin Heidelberg, pp. 617–629. ISBN: 978-3-540-22343-6. DOI: 10.1007/978-3-540-24855-2_70. URL: http://dx.doi.org/10.1007/978-3-540-24855-2_70.

Ryan, Conor, Atif Azad, et al. (2002). "No Coercion and No Prohibition, a Position Independent Encoding Scheme for Evolutionary Algorithms – The Chorus System". English. In: *Genetic Programming*. Ed. by JamesA. Foster et al. Vol. 2278. Lecture Notes in Computer Science. Springer Berlin Heidelberg, pp. 131–141. ISBN: 978-3-540-43378-1. DOI: 10.1007/3-540-45984-7_13. URL: http://dx.doi.org/10.1007/3-540-45984-7_13.

Ryan, Conor, Miguel Nicolau, and Michael O'Neill (2002). "Genetic Algorithms Using Grammatical Evolution". English. In: *Genetic Programming*. Ed. by JamesA. Foster et al. Vol. 2278. Lecture Notes in Computer Science. Springer Berlin Heidelberg, pp. 278–287. ISBN: 978-3-540-43378-1. DOI: 10.1007/3-540-45984-7_27. URL: http://dx.doi.org/10.1007/3-540-45984-7_27.

Steenstra, Joris (2015). *BS-PiGe Bachelorscriptie - Position Independend Grammatical Evolution*. URL: https://github.com/steenstra/BS-PiGe.