TU Chemnitz

Fakultät für Informatik

Stephan Heinich

Genetische Programmierung in C++

Wintersemester 2001/02

Proseminar: Genetische Programmierung
Proseminarleiter: Dipl.–Inf. J. Arnold

Inhalt

1. Beispiel: Symbolic Regression

Es wird eine Funktion, z.B. $x*x*x*x+x*x*x+x*x+x$, vorgegeben. Es sollen Funktionen, in Form von Genetischen Programmen (abgekürzt: GP), zufällig erzeugt werden. Diese GP werden über mehrere Generationen gekreuzt und mutiert. Das geschieht so lange, bis ein oder mehrere GP der vorgegebenen Funktion möglichst genau entsprechen. Die GP können in unserem Fall das Terminal x und die Funktionale (+ , – , * , %) benutzen. Es ist auch vorstellbar, dass man nicht nur ein Terminal sowie andere Funktionale benutzt. Die vorgegebene Funktion könnte man auch durch eine Reihe von Ergebniswerten eines Tests ersetzten. Das wäre sehr nützlich, wenn man zu den Ergebnissen eine Funktion sucht.

2. Klassendefinition

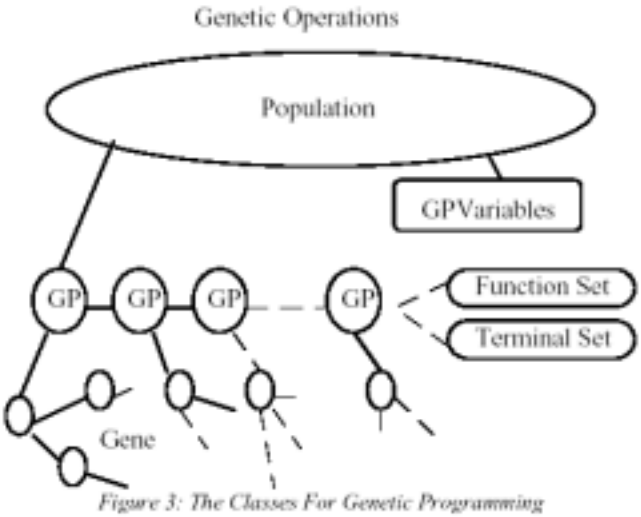


Figure 3: The Classes For Genetic Programming

Die Klassendefinition besteht aus der Population. Diese besteht aus den genetischen Programmen. Sie beinhaltet Variablen, die z.B. die Fitness der gesamten Population bestimmen.

```
class Population : public GPVariables
{
      GP *pgpHeader;
      unsigned long uliFitness,
      uliLenght,
      uliDepth;
};
```

Eine Unterstruktur der Population ist die `GPVariables`. Die `GPVariables` beinhaltet Variablen die den ganzen Aufbau und die Eigenschaften der Population festlegen.

Die Unterstruktur `GPVariables` enthält Variablen wie:

- Populationsgröße
- Anzahl der Generationen
- Anzahl der automatisch definierten Funktionen
- Erzeugungstyp der GP
- Anzahl der Bewertungen
- maximale Fitness
- maximale Tiefe der Erzeugung und der Kreuzung
- Anzahl der Mutationen

```
struct GPVariables
 {
       unsigned int PopulationSize,
       NumberOfGenerations,
       NumberOfADFs,
       CreationType,
       NumberOfEvaluations,
       MaximumFitness,
       MaximumDepthForCreation,
       MaximumDepthForCrossover,
       NumberToMutate;
       unsigned long MaximumSumFitness;
 };
```

Die Unterklassen der GP der Population stellen die eigentliche Population dar. Sie bestehen aus Bäumen von Genen. Wobei die Gene die so genannten Terminale bzw. Funktionale darstellen.

```
struct GP
{
       Gene **ppgHeader;
       unsigned int iFitness,
       iLength;
};


struct Gene
{
       unsigned int iValue;
       Gene *pgChild,
       *pgNext;
};
```

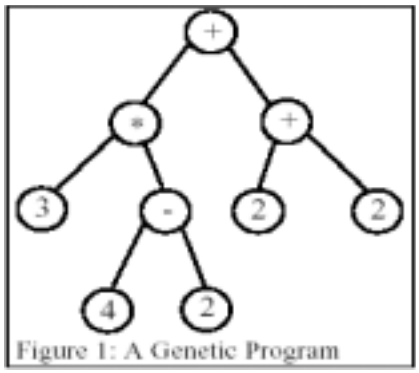


Figure 1: A Genetic Program

3. Erzeugung eines genetischen Programms

GP werden zufällig erzeugt. Die Knoten des Baums eines GP bestehen aus Funktionalen mit ebenso vielen Söhnen wie Argumenten. Dies ist in der oberen Abbildung zu erkennen. Es kann sein, dass man sehr viele GP produzieren muss. Dabei ist es ratsam, die GP unter bestimmten Kriterien zu erzeugen, um nicht an Grenzen des Speichers zu gelangen.

Man kann GP unter Berücksichtigung folgender 5 Erzeugungstypen erstellen:

3.1. Variable

- Ein GP wird maximal bis zu einer Größe bzw. Tiefe einer Struktur erzeugt.
- Das heißt, es werden nicht mehr Funktionale und Terminale, als vorgegeben, erzeugt.

3.2. Grow

- Es können so lange Funktionale zur Erzeugung verwendet werden, bis eine maximale Tiefe des Baums erreicht wurde, dann werden Terminale verwendet.

3.3. Ramped

- Die Population wird in mehrere kleine Blöcke aufgeteilt.
- Alle GP in einem Block haben die selbe Tiefe bzw. Größe.
- Ramped kann in zwei Arten erzeugt werden:
    - Ramped Variable: Die Blöcke werden mit Hilfe der Variable–Variante erzeugt.
    - Ramped Grow: Die Blöcke werden durch die Grow –Variante erzeugt.

3.4. Ramped Half and Half

- die Population wir zur Hälfte mittels Ramped Variable erzeugt und zur Hälfte mit Ramped Grow

4. Fitnessberechnung

Zur Fitnessberechnung legt man ein Maximum der Fitness (`MaxFitness`) von 1000 fest. Man kann nun mit den Zahlen 1 bis 10, als Funktionswerte, die GP berechnen. Zugleich wird die vorgegebene Funktion, in unserem Fall $f(x) = x*x*x*x + x*x*x + x*x + x$, mit den selben Zahlen berechnet. Die sich ergebenden Differenzen werden, zu einer Gesamtdifferenz, summiert. Sollte einmal eine Differenz einen Wert von 100 überschreiten, wird dieser auf 100 korrigiert. Die Gesamtdifferenz wird von dem festgelegten Maximum von 1000 abgezogen. Es sind demzufolge Werte von 0 bis 1000 möglich.

Zur näheren Erläuterung dient das folgende Beispiel. Ein zufällig ausgewähltes GP enthält folgende Funktion $g(x) = x*x*x*x + x*x*x + x$.

Funktionswerte	1	2	3	4	5	6	7	8	9	10
*x*x*x*x+x*x*x+x*x+x*	4	30	120	340	780	1554	2800	4680	7380	11110
*x*x*x*x+x*x*x+x*	3	26	111	324	755	1518	2751	4616	7299	11010
Differenzen	1	4	9	16	25	36	49	64	81	100

Gesamtdifferenz = 385, `Fitness` = `MaxFitness` – Gesamtdifferenz.
Damit ergibt sich eine Fitness für das GP von 615.
Quelltext für die Fitnessberechnung:

```
#define FITNESS float
#define FUNCTION( x ) x*x*x*x + x*x*x + x*x + x
float ques[10];
float answ[10];
float globalX;
GP *pgpGlobal;
void InitialiseGP()
{
    // F(main) = { *,+,-,% }
    // T(main) = { X }
    Translate = TranslateROOT;
    for ( int i = 0; i < 10; i++ )
    {
          ques[i] = (float)i;
          answ[i] = FUNCTION( (float)i );
    }
}

FITNESS TranslateROOT( Gene *pg )
{
  switch ( pg->iValue )
  {
   case 1: return Translate(pg->pgChild) * Translate(pg->pgChild->pgNext);
   case 2: return Translate(pg->pgChild) + Translate(pg->pgChild->pgNext);
   case 3: return Translate(pg->pgChild) - Translate(pg->pgChild->pgNext);
   case 4: return Divide( pg->pgChild );
   case 5: return globalX;
   default: return (unsigned int)(pg->iValue - 32768);
  }
}

unsigned int EvaluateFitness( GP *pgp, int Evals )
{
    FITNESS rawfitness = 0, diff = 0;
    pgpGlobal = pgp;
    Evals--;
    for ( int i = 0; i < 10; i++ )
    {
          float tempGPAnswer;
          globalX = ques[i];
          tempGPAnswer = Translate( ROOT );
          diff = fabs(answ[i] - tempGPAnswer);
          if ( diff > 100 ) diff = 100;
          rawfitness += diff;
    }
    return (1000 - (unsigned int)rawfitness );
}
```

5. Selektion

Mittels Selektion werden GP zum Kreuzen bzw. Mutieren ausgewählt. Dabei werden die Eltern zum Kreuzen und Kinder zum Speichern der Resultate ausgewählt. Bei der Mutation wird nur jeweils ein GP ausgewählt, welches dann mit der eigenen Mutante überschrieben wird.

Im Folgenden werden 3 Arten der Selektion vorgestellt:

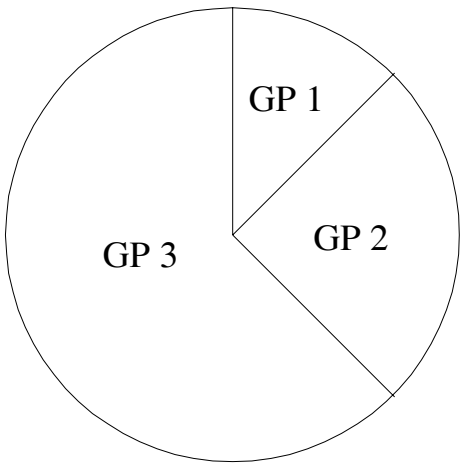
5.1. Fitness–proportionale Selektion

- Das Verfahren ähnelt dem eines Roulettespiels.
- Es wird die Summe aller Fitnesswerte der gesamten Population ermittelt, die sogenannte Gesamtfitness.
- Diese Summe beschreibt 100% des Roulettrades.
- Nun werden die prozentualen Anteile der Fitnesswerte zur Gesamtfitness ermittelt.
- Dieses Verfahren macht die guten GP wahrscheinlicher, gibt aber für die schlechten GP auch noch Chancen zur Selektion.
- Ein zufällig bestes GP wird über ein zufällig schlechtes (Inverses Verfahren) geschrieben.

Als Beispiel dient ein GP–System aus drei GP mit folgenden Fitnesswerten und den dazugehörigen prozentualen Anteilen, die sie später auf dem Rouletterad einnehmen:

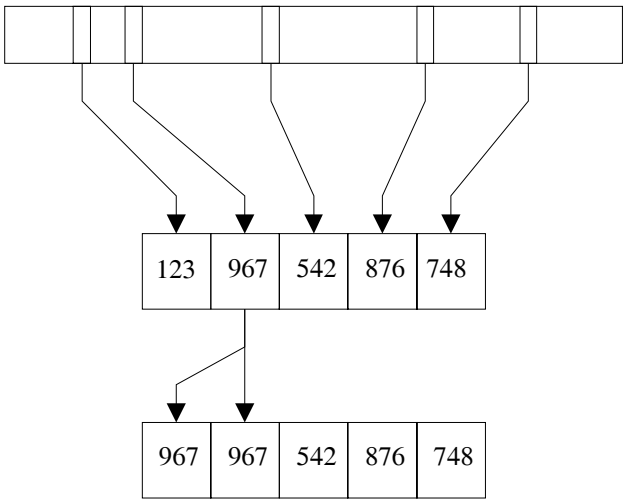
Genetische Programme	***Fitness***	***Prozentualer Anteil***
GP 1	188	12.50%
GP 2	375	25.00%
GP 3	937	62.50%
Gesamtfitness	1500	100.00%

Das Rouletterad ist dann wie im folgenden aufgeteilt:



5.2. Tournier–Selektion

- Aus der Population wird zufällig eine 5–er Gruppe gewählt.
- In der wird das schlechteste GP mit dem besten überschrieben (siehe folgendes Bild).
- Die Größe der Gruppe kann beliebig verändert werden.




```
#define TournamentSize 5
GP* Population::Select()
{
      int randgp = gp_rand() % PopulationSize;
      return (pgpHeader + randgp);
}

GP* Population::SelectBest()
{
      GP *pgpTournament[TournamentSize];
      GP *pgp;
      for ( int i = 0; i < TournamentSize; i++ )
            pgpTournament[i] = Select();
      pgp = pgpTournament[0];
      for ( i = 0; i < TournamentSize; i++ )
      {
            if ( pgp->iFitness < pgpTournament[i]->iFitness )
            pgp = pgpTournament[i];
      }
      return pgp;
}
```

5.3. Gruppen–Selektion

- Die Population ist in Gruppen (Demes) unterteilt.
- Diese Gruppen werden durch „Dämme“ von einander getrennt.
- Die GP können nur innerhalb der Gruppen interagieren, mit Ausnahme von den so genannten Überläufern (Migratoren).
- Diese können aber nur zur benachbarte Gruppen „auswandern“ (wie nachstehend abgebildet).

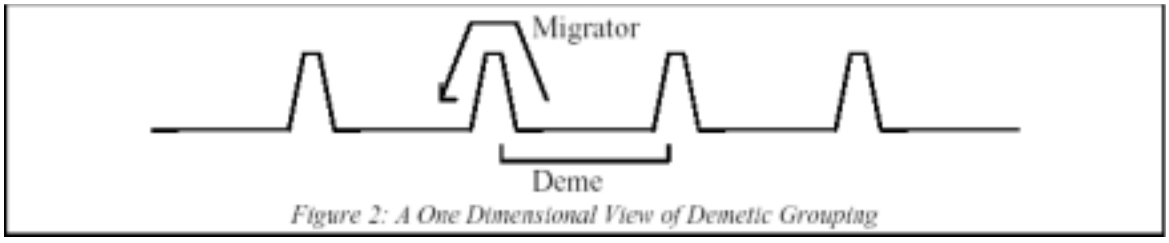


Figure 2: A One Dimensional View of Demetic Grouping

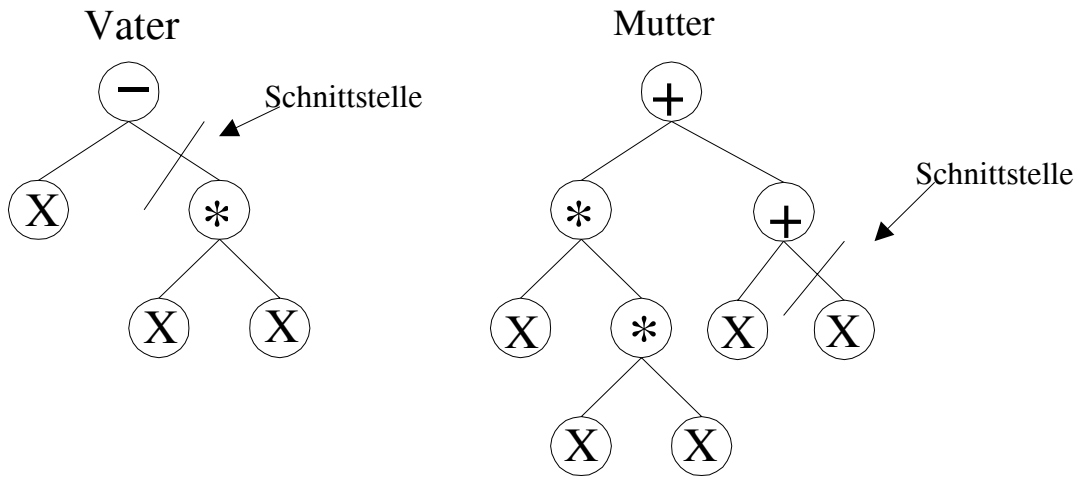
6. Kreuzung

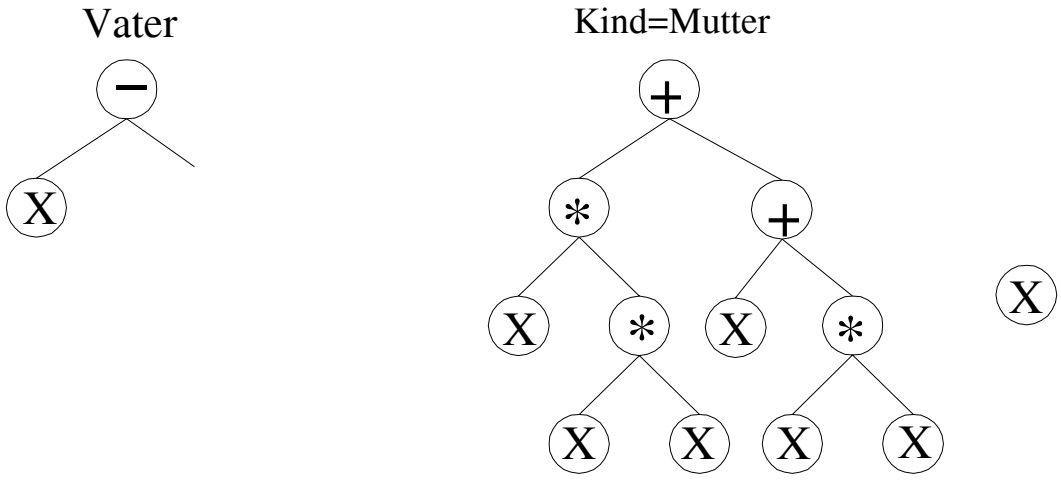
Das Kreuzen ist eine der wichtigsten Methoden in der Evolution von GP. Man kann damit die verschiedensten Ergebnisse erzielen, um besser an die anzustrebende Lösung zu gelangen. Meist kreuzt man zwei gute GP um ein eventuell noch besseres zu erhalten. Die schlechten können unter Umständen ebenso hilfreich sein. In Fällen, in denen sich die GP in einen inzestartigen Zustand entwickelt haben, in denen man einen Fehler nicht mehr herauskreuzen kann, sind solche schlechten GP sehr nützlich um die Entwicklungsvielfalt aufrecht zu erhalten. So kann es passieren, dass bei den guten GP, bezogen auf unser Beispiel, überhaupt kein „einzelnes“ x, sondern nur in Form von x^n mit $n > 1$ vorliegt.

Eine Kreuzung kann auf folgende 2 Arten erfolgen:

6.1. Subtree Crossover

- Es werden zwei Eltern ausgewählt.
- Nun wird vom Vater ein zufälliger Unterbaum „abgeschnitten“.
- Ein ebenso zufälliger Unterbaum der Mutter wird mit dem Unterbaum vom Vater ersetzt.
- Die veränderte Mutter wird dem Kind zugewiesen.
- Der Vater und der Unterbaum der Mutter werden im Anschluss zerstört.
- Die 2 folgenden Darstellungen sollen diese Methode veranschaulichen.





6.2. Ad Addendum

- Es wird ein GP zufällig ausgewählt.
- 2 oder mehr Knoten aus dem Baum des GP werden zufällig ausgewählt und untereinander ausgetauscht.
- Das ist eine sehr kompliziert zu programmierende Variante, da die Knoten mitunter die selben Unterbäume besitzen.

– das Subtree Crossover in C++:

```
void GP::Cross( GP *mum, GP *dad, int maxdepthforcrossover )
{
    Gene *cutchild, *cutdad;
    unsigned int MaxDepth = 0;
    int randtree = gp_rand() % RootandADF;
    do
    {
          Copy( mum );
          cutchild = (*(ppgHeader + randtree))->Choose();
          cutdad = (*( (dad->ppgHeader) + randtree ))->Choose();
          if ( cutchild->pgChild ) delete cutchild->pgChild;
          if ( cutdad->pgChild )
          {
                if ( !(cutchild->pgChild = new Gene(cutdad->pgChild)) )
                ExitSystem( "GP::Cross" );
          }
          else cutchild->pgChild = NULL;
          cutchild->iValue = cutdad->iValue;
          MaxDepth = 0;
          (*(ppgHeader + randtree))->Depth( 0, MaxDepth );
    } while ( MaxDepth > maxdepthforcrossover );
    Length();
}
```

7. Mutation

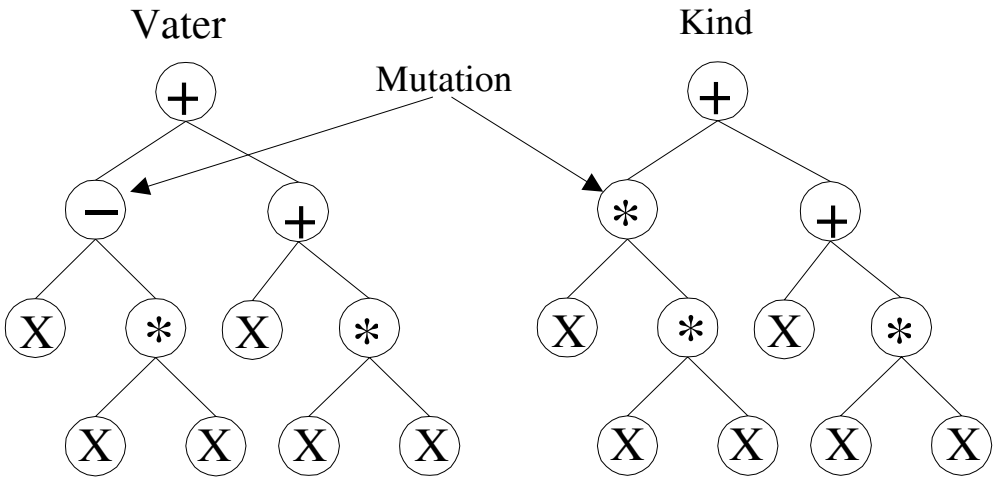
Die Mutation beschäftigt sich mit nur einem GP. Das zufällig ausgewählte GP wird durch Veränderung zu einem komplett neuen GP mutiert. Im Folgenden werden 2 Methoden der Mutation vorgestellt.

7.1. Mutation durch Genaustausch

- Hierbei handelt es sich um das Austauschen von einzelnen Genen und nicht von Unterbäumen.
- Es können nur Terminale mit Terminalen und Funktionale mit Funktionalen ausgetauscht werden.
- Dabei muss beachtet werden, dass bei den Funktionalen die Anzahl der Argumente übereinstimmen muss.
- So können z.B. nur '+' mit '–' oder '*' vertauscht werden und nicht mit einem Funktional, beispielsweise `abs()`, zum Bilden des Betrags einer Zahl.
- Da wir uns nur mit 2–argumentigen Funktionen beschäftigen, spielt das keine weitere Rolle.
- Wenn man weitere Terminale definiert, könnte auch ein x gegen ein y ausgetauscht werden.
- Bei dieser Methode werden keine neuen Bäume bzw. Unterbäume generiert.
- Im Folgenden ist eine solche Mutation, in der ein '–' zu einem '*' „mutiert wird, dargestellt.
- Aus einem GP mit der Funktion $f(x)=2*x$ entsteht die Funktion $g(x)=x*x*x+x*x+x$, dabei wurde nur ein Funktional verändert.

7.2. Mutation durch Schrumpfen

- Aus einem zufällig ausgewählten GP wird ein ebenso zufällig ausgewählter Unterbaum bestimmt.
- Dieser wird einfach über das GP geschrieben, dadurch schrumpft das GP.
- Die Methode ist teilweise hilfreich für sehr lange GP.
- Mit Ausnahmen funktioniert diese Methode nur mit ADFs (automatisch definierten Funktionen).



8. Zusammenfassung

In dem Referat wurde besprochen, wie speziell das GP−System der Symbolic Regression aufgebaut ist. Dabei wurde erklärt, wie genetische Programme erzeugt werden. Des Weiteren wurden drei Verfahren zur Selektion vorgestellt, von denen einige an darwinistische Modelle erinnern. Als Beispiel werden an dieser Stelle die Fitness−proportionale Selektion angeführt. Zur Erzeugung neuer Generationen werden Methoden verwendet, die versuchen natürliche Prozesse nachzuahmen. So besteht die Vererbung bei Lebewesen ebenfalls vorwiegend aus Kreuzung und Mutation. Diese werden durch äußere Einflüsse mitbestimmt. Ähnliche Einflüsse könnte man in einem GP−System implementieren, um der natürlichen Fortpflanzung noch näher zu kommen. Man kann sich auch vorstellen, dass man den Genen bestimmte Eigenschaften zuweist, wie zum Bsp. „rezessiv“ oder „dominant“. Es wären so eventuell schnellere evolutionäre Erfolge möglich. Selbst die Einsatzgebiete der GP−Systeme können sehr vielfältig sein. Es können bestimmte Bewegungsabläufe, z.B. von Robotern, durch genetische Programmierung vereinfacht bzw. verbessert werden.

9. Literaturangabe

− Adam Peter Fraser: *„Genetic Programming In C++“*, University of Salford, Cybernetics Research Institute, Technical Report 040, 1994