Evolutionsstrategien

Seminar Evolutionary Algorithms SS 2010

Cornelius Diekmann
Lehrstuhl Robotics and Embedded Systems
Veranstalter: Dipl.-Inf. Frank Sehnke
Fakultät für Informatik
Technische Universität München
Email: diekmann@in.tum.de
27. April 2010

I. Einleitung

Die Idee der Evolution, dass sich durch Weitergabe und Mutation von genetischen Merkmalen und durch natürliche Selektion komplexe Lebewesen entwickeln konnten, ist heutzutage wissenschaftlich unumstritten. Es stellt sich nun die Frage, ob sich dieses biologische Phänomen mathematisch modellieren lässt und somit auch in anderen Bereichen als der Biologie anwenden lässt. Durch heutige Rechenkapazitäten, die sich an nahezu jedem Arbeitsplatz finden lassen, könnten so die Mechanismen der Evolution – die sich über Milliarden von Jahren entwickelt haben – genutzt werden, um Probleme innerhalb kürzester Zeit zu lösen. Die Idee ist es, die Evolutionstheorie als mathematisches Optimierungsproblem auszuformulieren, um so komplexe Probleme zu lösen.

II. Grundidee

„Evolutionsstrategien sind heuristische Optimierungsverfahren und gehören zu den Evolutionären Algorithmen.“ [1] Die Idee ist, dass man einen Vektor von n Variablen hat, welche einen möglichst optimalen Wert annehmen sollen. Es handelt sich hierbei um ein iteratives Verfahren, wobei der Grenzwert der Iteration in den meisten Fällen die optimale Lösung darstellt. Evolutionsstrategien basieren auf dem Prinzip der starken Kausalität: „*Ähnliche Ursachen haben ähnliche Wirkungen*“ wie es [2] kurz und bündig formuliert. Auf dieser Grundlage lässt sich das Optimierungsproblem bildlich nun wie folgt darstellen: Die n zu optimierenden Variablen werden als n-dimensionales stetiges Qualitätsgebirge betrachtet, wobei der höchste Gipfel das Optimum ist. Wir setzen sogenannte Eltern als Repräsentanten einer möglichen Lösung darin aus. Danach erzeugen wir Nachfahren aus ihnen, indem wir einen Elter leicht abgeändert kopieren (Mutation) und wählen die besten Nachfahren als Eltern der nächsten Generation aus. Dies entspricht sozusagen dem virtuellen Bergsteigen. Dieser Prozess wird wiederholt, bis einer der Nachkommen den Gipfel erklommen hat.

III. Abschätzung der Dauer

Schon vor Beginn der Iteration lässt sich abschätzen, wie viele Generationen in Etwa benötigt werden, um beliebig nahe an das Optimum zu kommen. Die Abschätzung der Generationenzahl γ lässt sich wie folgt berechnen:

$$\gamma = \frac{4n}{c_{\mu,\lambda}^2} \ln\left(\frac{1}{\epsilon\sqrt{6}}\right)$$

wobei $c_{\mu,\lambda}$ den sogenannten Fortschrittsbeiwert darstellt und ϵ die maximal erlaubte relative Abweichung vom Optimum. $c_{\mu,\lambda}$ hängt von der Elternzahl μ und der Nachkommenzahl λ ab und liegt im Allgemeinen zwischen 1 und 3 . [2]

IV. Evolutionsstrategien

A. Algorithmus

Der generelle Ablauf einer Evolutionsstrategie lässt sich wie folgt beschreiben:

0) **Initialisierung:** Erzeuge Startwerte bzw. Eltern zufällig
1) **Selektion:** Wähle die fittesten Individuen als Eltern der aktuellen Generation aus

2) **Rekombination und Mutation:** erzeuge aus den Eltern mutierte Nachfahren
3) **Bewertung:** Berechne die Qualität der Nachfahren und markiere sie als potentielle Eltern der neuen Generation
4) solange Abbruchbedingung nicht erfüllt: gehe zu 1

B. Terminologie

$(\mu \dot{+} \lambda)$ ist die allgemeine Darstellung einfacher Evolutionsstrategien. Sie bedeutet, dass μ Eltern λ Nachkommen erzeugen. Dabei wird der Elter für jeden Nachkommen zufällig gewählt und von ihm eine mutierte Kopie erstellt. Es wird die Qualität oder Fitness der Nachkommen ermittelt. Bei einer „+“-Strategie werden die μ fittesten Kandidaten aus den Elten und den Nachkommen gewählt, welche die Eltern für die nächste Runde bestimmen. Bei einer „,“-Strategie werden nur die μ fittesten Nachkommen gewählt.

$(\mu \dot{+} \lambda)$-MSR ist eine Erweiterung der einfachen Evolutionsstrategien, MSR steht für Mutations-Schrittweiten-Regelung und besagt, dass die Nachkommen die Schrittweite δ ihrer Eltern übernehmen und dabei δ zusätzlich noch um einen zufälligen Faktor mutiert wird. Die Schrittweite besagt, wie stark die n Variablen der Eltern verändert werden, um einen Nachkommen zu erzeugen.

$[\mu' \dot{,} \lambda'(\mu \dot{+} \lambda)]$ bezeichnet eine geschachtelte Evolutionsstrategie in der μ' Elternpopulationen λ' Kindgenerationen erzeugen. Jede dieser Generationen verfolgt isoliert von den anderen Populationen eine $(\mu \dot{+} \lambda)$ Strategie.

Eine $(\mu/\rho \dot{+} \lambda)$ Strategie beinhaltet einen Mischungsmechanismus bei dem ein Nachkomme entsteht, indem ρ Eltern gemischt werden, um genau einem Nachkommen zu erzeugen. Hierbei handelt es sich im eine diskrete Mischung, bei der für jede der n Variablen durch Zufall entschieden wird, aus welchem der ρ Eltern der Wert entnommen wird. Bei einer $(\overline{\mu/\rho} \dot{+} \lambda)$ Strategie wird eine kontinuierliche Mischung verwendet, bei der alle Werte der Eltern gemittelt werden, um einen Nachkommen zu erzeugen. $(2/2, \lambda)$ entspricht also der sexuellen Fortpflanzung.

Bei einer $[\mu' \dot{,} \lambda'(\mu \dot{+} \lambda)^{\gamma}]$ werden die λ' Populationen für γ Generationen isoliert. Es entsteht die allgemeine Schreibweise einer $[\mu'/\rho' \dot{+} \lambda'(\mu/\rho \dot{+} \lambda)^{\gamma}]^{\gamma'}$ Evolutionsstrategie.

C. Schrittweite

Fortschrittsgeschwindigkeit und Mutationsschrittweite hängen eng zusammen. Es existiert nur ein schmaler Bereich, in dem – abhängig von der Mutation – ein nennenswerter Fortschritt zu finden ist. Dieser Bereich wird Evolutionsfenster genannt. Abbildung 1 stellt diesen Bereich grafisch dar. Der hellblaue Bereich ist das Evolutionsfenster, links davon ist die Mutation so gering, dass kein nennenswerter Fortschritt erzielt werden kann, rechts ist die Mutation so groß, dass wir sozusagen über das Ziel hinausgeschossen sind.

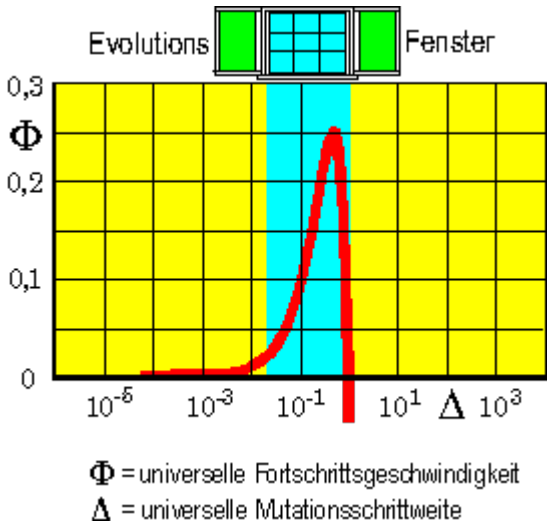

Abbildung 1. Evolutionsfenster, Quelle [3]

Durch eine Evolutionsstrategie mit Mutations-Schrittweiten-Regelung lässt sich die Fortschrittsgeschwindigkeit automatisch anpassen. Allerdings kann es auch manchmal nötig sein, dass Nachkommen mit zusätzlich erhöhter Schrittweite erzeugt werden müssen, denn eine Evolutionsstrategie verfolgt stets nur die lokal beste Strategie, kennt die globale Strategie jedoch nicht. Durch vergrößerte Schrittweiten kann die Strategie eine Art globale Weitsicht entwickeln.

D. Qualitätsfunktion

Durch eine Qualitätsfunktion lässt sich das anfangs erwähnte n-dimensionale Qualitätsgebirge modellieren, um so die Qualität bzw. Fitness der Nachkommen zu bewerten.

Die Idee ist ähnlich der Taylorreihe: Eine Funktion ist durch ein Polynom und das Restglied darstellbar, nach der starker Kausalität und, da wir uns nur im Evolutionsfenster befinden, kann die Entwicklung aber nach dem quadratischem Glied bereits abgebrochen werden. Somit ergibt sich nach [2] die Qualitätsfunktion

$$Q = Q_0 + \sum_{k=1}^{n} c_k y_k - \sum_{k=1}^{n} d_k y_k^2$$

mit c_k beliebig, d_k beliebig aber größer als 0. y_k sind die rechtwinkligen Achsen um den Elter, wobei das Koordinatensystem mit der Hauptachsentransformation gedreht wurde. Die Summen sind also ähnlich der Länge des Vektors der zu optimierenden n Variablen. Beispielsweise bilden $c_k \neq 0$ und $d_k = const$ eine Kreiskuppe.

Die Qualität der Nachkommen kann allerdings auch durch eine Computersimulation ermittelt werden. So kann zum Beispiel ein Problem optimiert werden, für das es mathematisch keine explizite Darstellung gibt und die Qualität jedes Prüflings numerisch abgeschätzt werden muss. Allerdings sollte die numerische Simulation dabei nur von einem Vektor von Fließkommazahlen als Parameter abhängen. Um nicht-reellwertige Probleme zu optimieren, ist eine Mapping-Funktion in die Reellen Zahlen nötig [4].

E. Mutation und Rekombination

Evolutionsstrategien versuchen biologische Prozesse zu kopieren. Sowohl bei der Rekombinationsstrategie der diskreten, als auch bei der kontinuierlichen Mischung, ist die Idee, dass mehrere Eltern einem Nachfahren ihre genetischen Merkmale mitgeben. Dabei bedeutet beispielsweise die Schreibweise $(2/2, \lambda)$, dass der Nachfahre genau die Hälfte seiner Merkmale von jedem Elter erhalten hat. Durch die Rekombination wird sichergestellt, dass sich der Varibalenvektor der Nachfahren zwischen dem seiner Eltern befindet und somit der Abstand der Nachfahren zu den Eltern kleiner ist, als der Abstand der Eltern zueinander. Es gibt noch weitere Rekombinationsstrategien wie das Flat Crossover, Discrete N-Point Crossover, ... welche in [4] beschrieben werden.

Mutation wird als der wichtigste Bestandteil einer Evolutionsstrategie betrachtet. Erst durch die sinnvolle Adaption der sogenannten Strategieparameter[1] wird aus der Nachahmung der Evolution eine Evolutionsstrategie.

Bei der *konstanten* oder auch *standard Mutation* wie sie in [4] beschrieben wird, wird keine Schrittweitenanpassung vorgenommen. Es wird nur eine Zufallszahl aus einem konstanten Intervall auf den Variablenvektor addiert. Da hierbei keine Strategieparameter zum Einsatz kommen, handelt es sich genau genommen nur um eine Art „evolutionäres Optimierungsverfahren“ aber keine Evolutionsstrategie.

Die $1/5$-Erfolgsregel – wie Prof. Rechenberg in [5] beschreibt – besagt, dass etwa jede fünfte Mutation eine Verbesserung gegenüber den Eltern darstellen sollte. Ist dies nicht der Fall, sollte die Schrittweite geändert werden. Diese Regel leitet sich ab, indem einmal die Qualitätsfunktion mit einer Kugel approximiert wurde und einmal mit einem Korridor. Um die maximale Fortschrittsgeschwindigkeit zu erreichen, ergaben sich aus beiden Modellen der Mittelwert $1/5$.

Bei einer $(\mu \dot{+} \lambda)$-MSR-Strategie mit *globaler Mutation* wird eine Schrittweite für den gesamten Variablenvektor verwendet. Dabei ist diese eine Schrittweite der einzige Strategieparameter. Die Nachkommen werden also in einem kreisförmigen Raum um den Elter platziert.

Hingegen wird bei einer $(\mu \dot{+} \lambda)$-MSR-Strategie mit *lokaler Mutation* für jede Dimension des Problems eine eigene Schrittweite mitgespeichert. Es existieren also n Strategieparameter und die Nachfahren werden in einem ellipsenförmigen Raum[2] um den Elter platziert.

Bei einer geschachtelten Evolutionsstrategie kann es oft sinnvoll sein, durch lokale Mutation der inneren Strategie (auf Individuenebene) ein lokales Maximum zu suchen und durch globale Mutation der gesamten Populationen das globale Maximum zu suchen. So kann zum Beispiel in einem gutwilligen Gebirge mit vielen Gipfeln operiert werden. Die Individuen erklimmen dabei jeweils ihr lokales

[1] von denen die Schrittweite aufgrund des Modells des Evolutionsfensters als der Wichtigste erscheint

[2] bzw. einem ellipsoidförmigen Raum für $n > 2$

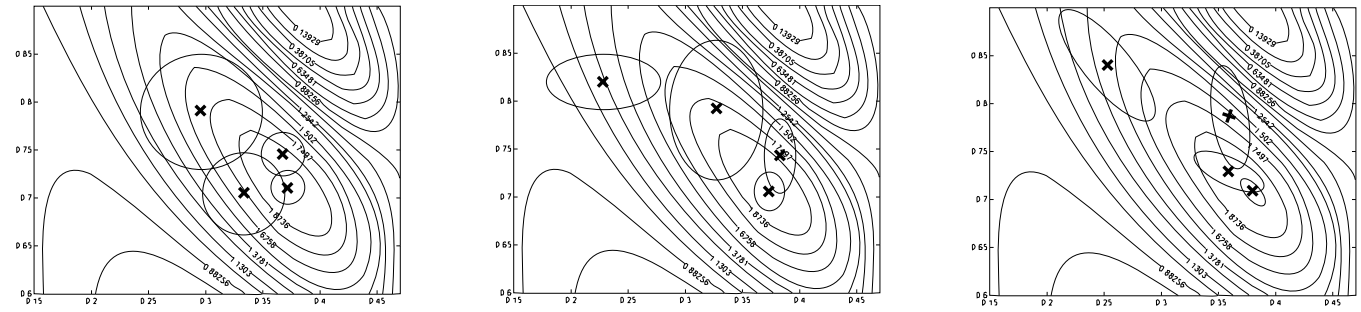

Abbildung 2. Mutationen im Vergleich. Quelle: [4]

Maximum, durch Mutation gesamter Populationen kann das globale Maximum gesucht werden.

Die *korrelierte Mutation* ist eine Erweiterung der lokalen Mutation bei der zusätzlich die Schrittweiten in Beziehung zueinander gesetzt werden. Es existieren genau $\frac{n(n-1)}{2}$ Winkel zwischen den n Koordinatenachsen der n-dimensionalen Problemstellung. Diese Winkel werden als zusätzliche Strategieparameter genutzt um den ellipsenförmigen Raum um den Elter, in dem die Nachkommen erzeugt werden, ideal zum Problem auszurichten. Die korrelierte Mutation wird von Hildebrand in [6] als „Mutationsart mit der größten Fähigkeit zur Anpassung an das Optimierproblem“ beschrieben.

Abbildung 2 zeigt die lokale, globale und korrelierte Mutation im Vergleich. Es lässt sich gut erkennen, wie sich die Ellipse der korrelierten Mutation wie ein direkter Pfad in Richtung Gipfel entwickelt.

Des Weiteren existiert unter Anderem noch die *Covarianz Matrix Adapatation (CMA)* welche den Rahmen dieses Papers sprengen würde.

F. Codebeispiel

Unter der Annahme, dass die Definitionen in Abbildung 3 bereits existieren, soll nun eine einfache $(5, 20)$-MSR Evolutionsstrategie mit globaler Mutation am Rechner getestet werden. Als Qualitätsfunktion soll hier $Q = \sum_{k=1}^{n} x_k - \sum_{k=1}^{n} x_k^2$ gewählt werden, welche ihr Maximum bei $x_k = 0,5$ erreicht. Abbildung 4 zeigt die Simulationsdurchführung. In den meisten Testläufen lag der absolute Fehler [3] in der Größenordnung $0,003$. Die Ausführungszeit der Simulation auf einem AMD Athlon X2 5200+ lag bei unter einer Sekunde. Abbildung 5 zeigt die Umsetzung der Evolutionsstrategie. Wenn man nun den Code wie in Abbildung 6 (Anhang) in eine Evolutionsstrategie mit lokaler Mutation ändert, wächst der absolute Fehler rasant auf die Größenordnung $171,877$ an. Das arithmetische Mittel des Ergebnisvektors ist in den meisten Komponenten nahe am Optimum $0,5$, jedoch weichen einige Komponenten sehr stark davon ab. Damit der absolute Fehler auf eine Größenordnung von $0,0135$ verkleinert werden kann, sind über 5000 Generationen und eine Ausführungszeit von mehr als zwei Sekunden nötig. Der absolute Fehler verringert sich auf $3,2562e-7$, wenn zusätzlich noch eine diskrete Mischung, wie in Abbildung 7 (Anhang), verwendet wird.

Wenn die Problemstellung auf 3000 Dimensionen erhöht wird, liegt die benötigte Rechenzeit der einfachen $(5, 20)$-MSR Strategie mit globaler Mutation für 5000 Generationen bereits bei über zwei Minuten. Das Ergebnis ist mit einem absoluten Fehler von ca. 29643 kaum verwertbar. Bei der erweiterten Strategie mit lokaler Mutation und diskreter Mischung lag die Ausführungszeit bereits bei über drei Minuten und der Fehler explodierte auf $5.18173e+12$. Dies zeigt, wie sehr das Ergebnis von der richtigen Evolutionsstrategie abhängt und wie viel Rechenzeit für vernünftige Ergebnisse im hochdimensionalen Raum nötig wäre.

[3] absoluter Fehler: $\sum_{k=0}^{n-1} |x_k - 0.5|$

```
// die zu optimierenden Varibalen werden im Typ Variables gespeichert
typedef QVector<double>  Variables;

// gleichverteilte [0,1) Zufallszahl
#define RAND_0_1 (rand() / static_cast<double>( RAND_MAX+1 ))

// normalverteilte (0,1/sqrt(x.size)) Zufallszahl
#define RAND_0_X (sqrt(-2*log(1-RAND_0_1)/length)*sin(6.2832*RAND_0_1))

// eine simple my lambda Evolutionsstrategie mit dem mehrdimensinalen Vektor x als Anfangswerte
QVector<Variables> simple_evo_strat(int my, int lambda, QVector<Variables> x, int generations);

// Die Funktion um die Qualitaet der Variablen zu bestimmen
double quality(Variables x);

// kontinuierliche Mischung aller Varibalen, zu Einer - durch das arithmetische Mittel
Variables arith_average(QVector<Variables> x);
```

Abbildung 3. C++ mit Qt Bibliothek: defs.h

```
#include "header.cpp"


int main(int argc, char *argv[])
{
    QCoreApplication a(argc, argv);

    // initialisiere Pseudo-Randomgeneartor mit Sekunden seit Mitternacht
    QTime midnight(0, 0, 0);
    srand(midnight.secsTo(QTime::currentTime()));

    printf("Beginn der Evolution");

    // erzeuge 30-dimensionale Testvektoren mit dem Startwert 42
    QVector<Variables> initial_values(5);
    initial_values.fill(Variables(30, 42));

    // wende eine (5,20) Evolutionsstrategie mit 500 Generationen an
    QVector<Variables> output = simple_evo_strat(5, 20, initial_values, 500);
    qDebug() << arith_average(output);

    return a.exec();
}
```

Abbildung 4. C++ mit Qt Bibliothek: main.cpp

V. Probleme

Ein generelles Problem bei Evolutionsstrategien stellt der sogenannte Fluch der Dimensionen dar. Im n-dimensionalen Raum ist die Wahrscheinlichkeit, dass ein Nachkomme besser wird als sein Elter, sehr gering und stark von der Mutationsschrittweite abhängig. Allerdings existiert laut [1] kein exaktes Verfahren um die richtige Schrittweite zu wählen. Des Weiteren wird im hochdimensionalen Raum sehr viel Rechenzeit benötigt.

Bei Evolutionsstrategien handelt es sich grundsätzlich um lokale Verfahren, die einer Gradientenstrategie gleichen. Hinsichtlich Konvergenzgeschwindigkeit und ob der errechnete Grenzwert das globale Optimum darstellt, lassen sich oft keine Aussagen treffen.

Bei der Implementierung werden viele normalverteilte Zufallszahlen benötigt. Die Standardbibliotheken gängiger Programmiersprachen verfügen oft jedoch nicht über Pseudozufallszahlengeneratoren

```
QVector<Variables> simple_evo_strat(int my, int lambda, QVector<Variables> x, int generations){
    assert(x.size() == my);
    // Laenge des Variablenvektors
    int length = x[0].size();

    // X: zu optimierende Variablen
    // _n Nachkomme
    // _e Eltern
    // _b Besten
    Variables X_n(length);
    QVector<Variables> X_e(my, Variables(length)), X_b(my, Variables(length));

    // S: Schrittweiten, Q: Qualitaet
    double S_n, Q_n;
    QVector<double> S_e(my), S_b(my), Q_b(my);

    // A: Schrittweitenanpassung
    double A(1.3);

    X_e = x;
    S_e.fill(A);


    for(int g = 0; g < generations; ++g){

        // Erzeugung der Nachfahren
        Q_b.fill(-DBL_MAX);
        for(int i = 0; i < lambda; ++i){

            // Erzeugung eines Nachfolgers
            int elter = (int)(my * RAND_0_1);
            S_n = S_e[elter]*pow(A, 2*floor(RAND_0_1+0.5)-1);
            for(int j = 0; j < length; ++j){
                X_n[j] = X_e[elter][j] + S_n*RAND_0_X;
            }

            // Ermittlung des momentan schlechtesten Nachfolgers
            double worst = Q_b[0];
            int worst_index = 0;
            for(int j = 0; j < Q_b.size(); ++j){
                if(Q_b[j] < worst){
                    worst_index = j;
                    worst = Q_b[j];
                }
            }

            Q_n = quality(X_n);
            if(Q_n > worst){
                // Den schlechtesten Nachfolger mit dem aktuellen ersetzten
                S_b[worst_index] = S_n;
                Q_b[worst_index] = Q_n;
                X_b[worst_index] = X_n;
            }
        }

        // Die besten Nachfahren werden die neuen Eltern
        X_e = X_b;
        S_e = S_b;
    }



    return X_e;
}
```

Abbildung 5. C++ mit Qt Bibliothek: simple_evo_strat.cpp

ausreichender Güte oder leiden unter Effizienzproblemen.

VI. Zusammenfassung und Ausblick

Wir haben die grundlegenden Ideen der Evolutionsstrategien theoretisch kennengelernt und an einem Beispiel praktisch getestet. Es zeigt sich, dass Evolutionsstrategien ein mächtiges Werkzeug sein können, um im hochdimensionalen Raum zu Optimieren. Allerdings zeigt sich auch, wie sehr das Ergebnis mit der verwendeten Strategie und den unterschiedlichen Parametern schwanken kann. Ein Gefühl des Evolutionsstrategen um den passenden Algorithmus für das entsprechende Problem zu wählen, ist also stets von Nöten.

Literatur

[1] wikipedia, "Evolutionsstrategie," 03 2010. [Online]. Available: http://de.wikipedia.org/wiki/Evolutionsstrategie

[2] I. Rechenberg, *Evolutionsstrategie '94*. frommann-holzboog, 1994.

[3] Prof. Dr. Ingo Rechenberg, "Das Evolutionsfenster," 03 2010. [Online]. Available: http://www.bionik.tu-berlin.de/institut/s2evfen.html

[4] F. Streichert and H. Ulmer, "JavaEvA - a java framework for evolutionary algorithms," Centre for Bioinformatics Tübingen, University of Tübingen, Technical Report WSI-2005-06, 2005. [Online]. Available: http://w210.ub.uni-tuebingen.de/dbt/volltexte/2005/1702/

[5] I. Rechenberg, *Evolutionsstrategie. Optimierung technischer Systeme nach Prinzipien der biologischen Evolution*. frommann-holzboog, 1973.

[6] L. Hildebrand, "Schiefe-Ein Ansatz zur Erweiterung von Evolutionsstrategien um gerichtete Mutation," *Universit
ät Dortmund, Forschungsbericht*, vol. 639, 1997.

VII. Anhang

```
QVector<Variables> simple_evo_strat(int my, int lambda, QVector<Variables> x, int generations){
    assert(x.size() == my);
    // Laenge des Variablenvektors
    int length = x[0].size();

    // X: zu optimierende Variablen
    // _n Nachkomme
    // _e Eltern
    // _b Besten
    Variables X_n(length);
    QVector<Variables> X_e(my, Variables(length)), X_b(my, Variables(length));

    // A: Schrittweitenanpassung
    double A(1.3);

    // S: Schrittweiten, Q: Qualitaet
    QVector<double> S_n(length);
    double Q_n;
    QVector< QVector<double> > S_e(my, QVector<double>(length, A)), S_b(my, QVector<double>(
        length));
    QVector<double> Q_b(my);

    // S_e bereits mit A vorbelegt
    X_e = x;


    for(int g = 0; g < generations; ++g){

        // Erzeugung der Nachfahren
        Q_b.fill(-DBL_MAX);
        for(int i = 0; i < lambda; ++i){

            // Erzeugung eines Nachfolgers
            int elter = (int)(my * RAND_0_1);
            for(int j = 0; j < length; ++j){
                // Schrittweiten
                double s_anp = pow(A, 2*floor(RAND_0_1+0.5)-1);
                S_n[j] = S_e[elter][j]*s_anp;
            }
            for(int j = 0; j < length; ++j){
                // Variablen
                X_n[j] = X_e[elter][j] + S_n[j]*RAND_0_X;
            }

            // Ermittlung des momentan schlechtesten Nachfolgers
            double worst = Q_b[0];
            int worst_index = 0;
            for(int j = 0; j < Q_b.size(); ++j){
                if(Q_b[j] < worst){
                    worst_index = j;
                    worst = Q_b[j];
                }
            }

            Q_n = quality(X_n);
            if(Q_n > worst){
                // Den schlechtesten Nachfolger mit dem aktuellen ersetzten
                S_b[worst_index] = S_n;
                Q_b[worst_index] = Q_n;
                X_b[worst_index] = X_n;
            }
        }

        // Die besten Nachfahren werden die neuen Eltern
        X_e = X_b;
        S_e = S_b;
    }



    return X_e;
}
```

Abbildung 6. C++ mit Qt Bibliothek: simple_evo_strat_lokaleMSR.cpp

```
QVector<Variables> simple_evo_strat(int my, int lambda, QVector<Variables> x, int generations){
    assert(x.size() == my);
    // Laenge des Variablenvektors
    int length = x[0].size();

    // X: zu optimierende Variablen
    // _n Nachkomme
    // _e Eltern
    // _b Besten
    Variables X_n(length);
    QVector<Variables> X_e(my, Variables(length)), X_b(my, Variables(length));

    // A: Schrittweitenanpassung
    double A(1.3);

    // S: Schrittweiten, Q: Qualitaet
    QVector<double> S_n(length);
    double Q_n;
    QVector< QVector<double> > S_e(my, QVector<double>(length, A)), S_b(my, QVector<double>(
        length));
    QVector<double> Q_b(my);

    // S_e bereits mit A vorbelegt
    X_e = x;


    for(int g = 0; g < generations; ++g){

        // Erzeugung der Nachfahren
        Q_b.fill(-DBL_MAX);
        for(int i = 0; i < lambda; ++i){

            // Erzeugung eines Nachfolgers
            for(int j = 0; j < length; ++j){
                int elter = (int)(my * RAND_0_1);
                // Schrittweiten
                double s_anp = pow(A, 2*floor(RAND_0_1+0.5)-1);
                S_n[j] = S_e[elter][j]*s_anp;

                // Variablen
                X_n[j] = X_e[elter][j] + S_n[j]*RAND_0_X;
            }

            // Ermittlung des momentan schlechtesten Nachfolgers
            double worst = Q_b[0];
            int worst_index = 0;
            for(int j = 0; j < Q_b.size(); ++j){
                if(Q_b[j] < worst){
                    worst_index = j;
                    worst = Q_b[j];
                }
            }

            Q_n = quality(X_n);
            if(Q_n > worst){
                // Den schlechtesten Nachfolger mit dem aktuellen ersetzten
                S_b[worst_index] = S_n;
                Q_b[worst_index] = Q_n;
                X_b[worst_index] = X_n;
            }
        }

        // Die besten Nachfahren werden die neuen Eltern
        X_e = X_b;
        S_e = S_b;
    }



    return X_e;
}
```

Abbildung 7. C++ mit Qt Bibliothek: simple_evo_strat_lokaleMSR_diskreteMischung.cpp