Genetische Algorithmen in Komposition und Computermusik

Georg Holzmann
Dez. 2003

Seminararbeit für
Computermusik 1

Institut für Elektronische Musik und Akustik, Graz

Einleitung

Meist werden komplexe Systeme von Algorithmen in der algorithmischen Komposition verwendet. Dadurch entsteht eine Vielfalt an Parameter, die die intuitive Steuerung solcher Systeme erschweren.
Mit Hilfe von Interaktiven Genetischen Algorithmen (IGA) kann man Variationen dieser unzähligen Parameter nach eigenen ästhetischen Vorstellungen „züchten“, ohne ein Wissen über die darunter liegende Struktur haben zu müssen und hat trotzdem noch ein hohes Maß an Kontrolle.

Genetische Algorithmen (GA) sind im Prinzip eine Klasse von sehr leistungsstarken Methoden für Such- und Optimierungsprobleme. Sie basieren auf dem natürlichen Vorbild der Evolution. Über viele Generationen entwickeln sich Populationen in der Natur nach den Prinzipien der natürlichen Selektion („Überleben des Stärkeren“), Fortpflanzung und Mutation, welche zuerst von Charles Darwin in seinem Buch „The Origin of Species“ postuliert wurden.
Indem diese Prinzipien der Natur abgeschaut und imitiert werden, können genetische Algorithmen künstliche Populationen generieren und einer Evolution unterziehen, deren einziges Ziel es ist, die ihnen gestellten Probleme möglichst optimal zu lösen. [1]
Ein weiterer Vorteil der GA ist, dass man kein problemspezifisches Wissen über die von ihnen zu lösende Aufgabe braucht, man muss lediglich in der Lage sein, Zwischenergebnisse der GA bewerten zu können (Selektion).
Aus diesem Grund werden sie vor allem in solchen Aufgabengebieten eingesetzt, welche entweder noch nicht gut verstanden werden oder deren komplette Modellierung aus mathematischen oder rechnerischen Gründen unmöglich ist.

Geschichte der Genetischen Algorithmen

Die Fundamente der GA wurden in den 60er Jahren in den USA von John Holland gelegt und 1975 veröffentlicht. Jedoch erst ca. 10 Jahre später setzte sich langsam diese neue Art der Problemlösung durch und mit David E. Goldbergs Buch „Genetic Algorithms in Search, Optimization, and Machine Learning“ wurde dieses Thema einer breiten Öffentlichkeit zugänglich gemacht.[2]

Technisches Prinzip

Vorbild Natur

In einer natürlichen Umgebung treten einzelne Individuen einer Spezies in einen Konkurrenzkampf um Ressourcen wie Futter, Wasser und Schutz. Diesen Konkurrenzkampf fechten sie sowohl mit Vertretern anderer Spezies als auch untereinander aus. Zusätzlich müssen sie für Nachwuchs sorgen, das heißt, sie müssen sich nach einem geeigneten Partner umschauen und diesen dann erfolgreich umwerben. Diejenigen Vertreter einer Spezies, die auf den Gebieten des Überlebens und der Partnerwerbung am erfolgreichsten agieren, werden mit hoher Wahrscheinlichkeit eine größere Anzahl an Nachkommen haben als ihre weniger

[1] Nach http://www.chevreux.org/diplom/node32.html
[2] Nach http://www.chevreux.org/diplom/node32.html

glücklichen Artgenossen, welche z.B. einem Räuber zum Opfer gefallen sind oder für Paarungspartner unattraktiv sind.
Auf diese Weise werden Gene von erfolgreichen Individuen im Durchschnitt öfter vererbt als die der weniger erfolgreichen (Selektion). Mit jeder Generation werden diese Gene also weiter verbreitet. In einigen Fällen können verschiedene Genkombinationen bei einer Paarung oder nach Mutation entstehen, die zusammen wiederum ihren Trägern eine bessere Adaption an dessen Umwelt ermöglichen. [3]

Lebensraum Computer

GA befinden sich im „Lebensraum" Computer. In diesem arbeiten sie mit Populationen von Individuen, wobei jedes Individuum Variablen zur Lösung eines vorgegebenen Problems beinhaltet.
Die Bedingung für das Überleben stellen die Objective Score bzw. Fitnessfunktionen, die jedem einzelnen Individuum aufgrund seiner Leistung beim Lösen dieses Problems eine Fitness zuordnen.
Den gut angepassten Individuen wird dann Gelegenheit gegeben, Nachkommen mit anderen zu zeugen, um sich damit fortzupflanzen. Bei dieser Fortpflanzung werden die natürlichen Mechanismen des Crossovers (Paarung) und der Mutation angewandt, so dass die Nachkommen Gene von jedem Elternteil bekommen und zusätzlich noch ein Unsicherheitsfaktor eingebaut ist.

Einfacher Pseudoalgorithmus für GA

1) Definition der benötigten Parameter (des Genoms)

2) Initialisierung einer Ausgangspopulation (z.B. zufällig entstandene Individuen)

3) Ausrechnen des Objective Score bzw. der Fitness der Individuen

4) Selektion der Eltern für die Nachfolgegeneration und Crossover (Paarung)

5) Mutation der entstandenen Nachkommen

6) Einfügen der Nachkommen in die Ausgangspopulation

7) Solange Abbruchkriterium nicht erfüllt, zurück zu Punkt 3

8) Ausgabe der Endergebnisse

Je nach Implementation der GA können einzelne Schritte wegfallen, hinzukommen oder etwas früher oder später ausgeführt werden.

Vier Grundvarianten von Genetischen Algorithmen[4]

1) simple GA

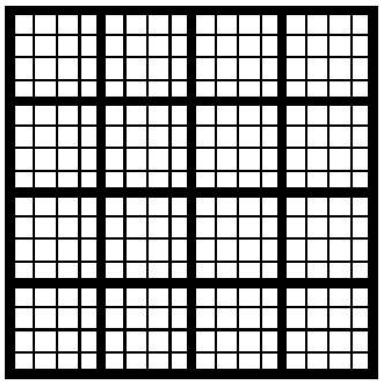
[3] Nach http://www.chevreux.org/diplom/node32.html
[4] Einteilung nach GAlib für C++: http://lancet.mit.edu/ga/

Es gibt keine überlappenden Generationen, d.h. die Nachkommen ersetzen die Elterngeneration vollständig.

2) steady-state GA
Die Generationen überlappen sich, man kann wählen, wie viel der Gesamtpopulation jeweils von der neuen Generation ersetzt wird.

3) incremental GA
In jeder Generation gibt es nur ein oder zwei Nachkommen. Wichtig ist dabei, wie sich diese in die Gesamtpopulation einbringen bzw. wen sie ersetzen (z.B. das Individuum mit dem niedrigsten Fitness Faktor, die Eltern, zufällig gewählte Individuen, …).

4) deme GA



Dieser GA entwickelt mehrere getrennte Populationen parallel und in jeder Generation können Individuen mit einer bestimmten Wahrscheinlichkeit von einer Population in eine andere wandern.

Im Bild links[5] sind es z.B. 256 einzelne Individuen, die auf 16 „Inseln" zu je 16 „Einwohnern" leben.
Die Individuen können mit einer vorgegebenen Wahrscheinlichkeit in eine andere Insel wandern („Migrationswahrscheinlichkeit").

Genom bzw. Repräsentation

Das Genom besteht, wie auch in der Natur, aus einzelnen Genen. Ein Gen ist ein Funktionsblock, der für einen bestimmten Parameter zuständig ist.
Ein wichtiger Punkt ist, eine geeignete Datenstruktur zu wählen, die das Problem am besten darstellen kann.
Diese Repräsentation soll so einfach als möglich gewählt werden, aber muss auch in der Lage sein alle möglichen Erscheinungsformen der Lösung auszudrücken. Weiters sollte diese Repräsentation alle unmöglichen Lösungen ausschließen.
Die Datentypen können integers, floats, arrays, Listen, bit strings, trees, ..sein, es können auch mehrere Datentypen in einem Genom vorkommen (dann muss allerdings auch das Crossover bestimmt geregelt werden!).

Z.B. können alle Parameter eines bestimmten Kompositionsalgorithmus als eine Liste oder als ein array aus floats dargestellt werden. Das gesamte Array entspricht nun dem Genom und ein einzelner Parameter einem Gen. Verwandte Parameter werden nun nebeneinander im Genom platziert (z.B. alle Filterparameter, oder alle Parameter, die einen bestimmten Oszillator kontrollieren).
Die Abbildung der Parameter auf dem Genom kann nun mit verschiedenen Umrechnungskurven erfolgen, um die Wahrscheinlichkeit des gewünschten Ergebnisses zu erhöhen, aber trotzdem noch die Möglichkeit von „Ausreißern" zuzulassen. Z.B. man will, dass ein LFO die Tonhöhe ein wenig moduliert. Zu viel Modulation soll verhindert werden, es soll aber auch nicht die Möglichkeit von vielleicht gut klingenden Ausreißern ausgeschlossen

[5] Bild von http://timara.con.oberlin.edu/~gnelson/papers/morph93/morph93.pdf, Seite 6

werden. Deshalb würden sich in diesem Fall nicht lineare Umrechnungskurven anbieten (exponentiell, kubisch, logarithmisch, …).

Initialisation

Die Initialisation determiniert, wie das Genom initialisiert wird. Z.B. es werden bestimmte vorher gespeicherte Zustände geladen, oder es wird zufällig initialisiert, nach bestimmten math. Formeln, …
Die Initialisation kann für jedes Individuum verschieden sein, oder auch für die ganze Population gelten.

Objective Function und Fitness Scaling

Die Objective Function bewertet die Genome und es entsteht dadurch ein Fitness Scaling. Wichtig ist der Unterschied zwischen dem Objective Score und Fitness Score. Der Fitness Score kann z.B. eine lineare Skala sein, die aus dem Objective Score abgeleitet wird.
Bei ästhetischen Objekten wie Bildern, Sounds, Kompositionen ist es natürlich sehr schwierig oder auch nicht beabsichtigt, eine automatisierte Zuordnung der Fitness zu finden. Deshalb kam es zur Entwicklung der interaktiven GA (IGA), bei denen der Komponist oder Performer selbst die Bewertung einer bestimmten Population in realtime (oder auch nicht) vornehmen kann.

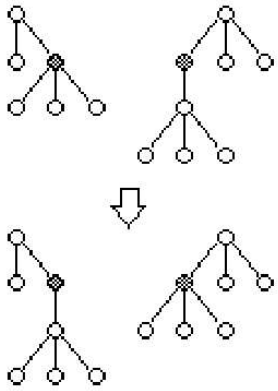
Selektion

Ein Selektionsschema definiert die Art und Weise, mit der die Eltern der Nachfolgegeneration bestimmt werden.
Durch das Fitness Scaling ergibt sich für jedes Individuum eine bestimmte Fortpflanzungswahrscheinlichkeit.
Ist die Selektion von Eltern einzig und allein von deren Fitness abhängig und werden nur die besten Individuen für die Fortpflanzung in Betracht gezogen, so wird dies als harte Selektion bezeichnet. Im Gegensatz dazu steht die weiche Selektion, welche auch weniger fitten Individuen eine Reproduktionschance gibt.

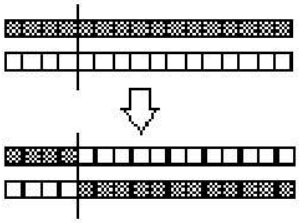
Crossover

Als Crossover bezeichnet man die Generierung eines oder mehrerer Nachkommen von zwei oder mehr Eltern.
Im einfachsten Fall werden zwei Genome an einer jeweils gleichen Stelle gebrochen und die entstehenden Teilstücke ausgetauscht (one-point crossover). Weitere Möglichkeiten bestehen zum Beispiel aus mehreren solcher Bruchstellen (n-point crossover). Die Möglichkeiten sind quasi unbegrenzt, und dieser Operator wird typischerweise sehr oft angewandt.
Zu beachten ist, dass der Crossover Operator für verschiedene Datentypen verschiedene Gestalten annehmen muss. Außerdem können die einzelnen Gene (Parameter) ihre Position im Genom behalten und sich nur mit korrespondierenden Genen vermischen, oder es kann sich das gesamte Genom vermischen.
Es ist natürlich auch möglich, bestimmte Gene absichtlich unverändert weiter zu geben (wenn einem z.B. eine Klangfarbe gefällt, aber der Rest soll sich weiterentwickeln).

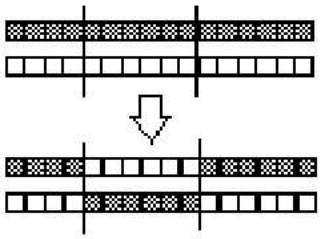
Crossover-Varianten:[6]



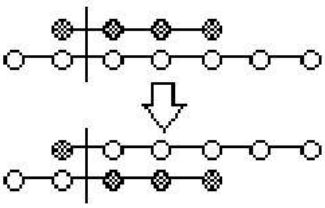
tree one point crossover



array one point crossover



array two point crossover



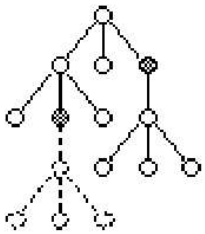
list one point crossover

Mutation

Die Mutation soll neues genetisches Material bringen, das zu neuen Lösungen führen kann. Sie ist natürlich wieder für jeden Datentyp unterschiedlich definiert. Z.B. bei einem Genom, das nur aus Bits besteht, kippen gewisse Bits mit einer bestimmten Wahrscheinlichkeit.
Man kann natürlich auch verschiedene Mutationsmechanismen für ein Problem verwenden.
Wie Crossover hat auch die Mutation sowohl destruktive als auch konstruktive Effekte auf eine Population. Die Mutationswahrscheinlichkeit sollte allerdings nicht zu hoch gesetzt werden, da sonst der GA in eine Zufallssuche abgleitet.

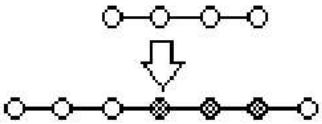
Mutations-Varianten:[7]

[6] Bilder von GAlib für C++: http://lancet.mit.edu/ga/
[7] Bilder von GAlib für C++: http://lancet.mit.edu/ga/



tree sub-tree swap mutation

list destructive mutation



list generative mutation

Suchraum: Exploration und Exploitation

Der Suchraum eines Problems kann als hyperdimensionale Ergebnislandschaft angesehen werden, mit mehreren Optima und Minima (bzw. gute und schlechte Lösungen).
Als Exploitation wird der Prozess bezeichnet, bei dem Information von früher besuchten Punkten im Suchraum für die Bestimmung der nächsten zu untersuchenden Punkte benutzt wird. Ein solches Verfahren eignet sich also gut, um ein lokales Optimum zu bestimmen.
Im Gegensatz dazu steht die Exploration. Hier werden Sprünge ins „kalte Wasser“ unbekannter Regionen gewagt, zu denen keine Information über die Güte zur Verfügung steht.
Aufgabenstellungen, deren Lösung viele lokale (Sub-) Optima enthält, können oftmals nur unter Zuhilfenahme von Exploration gelöst werden.
Entscheidend für den Erfolg eines Genetischen Algorithmus ist also die richtige Mischung aus Exploration und Exploitation. Dazu dienen die beiden Operatoren Crossover - mit explorativem und exploitativem Charakter - und Mutation, mit rein explorativem Anteil.

Musikalische Anwendungen

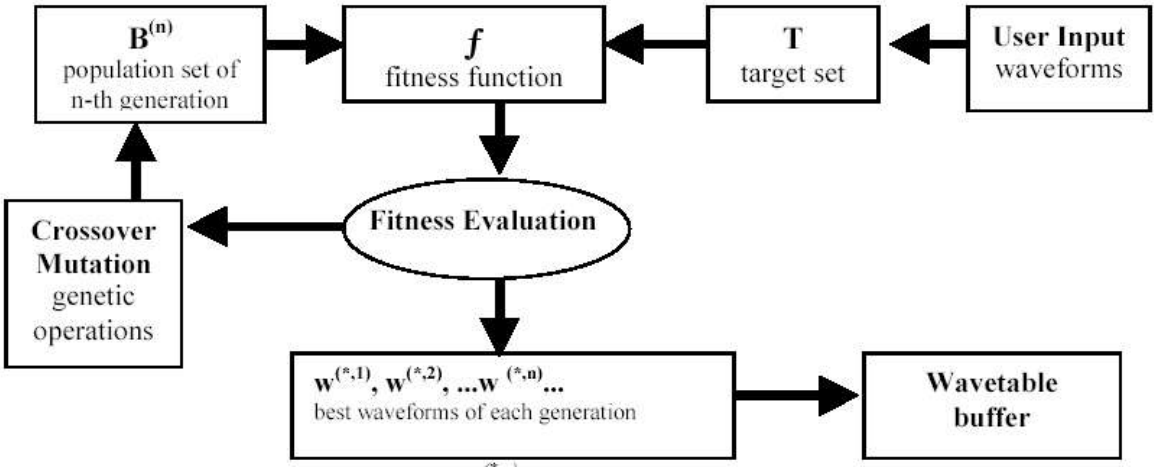
GenJam
(siehe http://gsd.ime.usp.br/sbcm/2000/papers/moroni.pdf)

GenJam von Biles, 1994, ist ein auf IGA (Interaktive Genetische Algorithmen) basierendes Programm, mit dem man dem Computer Jazz-Soli beibringen kann.
In GenJam gibt es zwei Arten von Individuen: Takte und Phrasen. Ein Phrasenindividuum besteht aus vier Taktindividuen. Diese beiden Individuen bilden also eine Hierarchie einer melodischen Struktur und sind sozusagen der Speicher an melodischen Ideen.
Das Programm improvisiert nun, indem es einen Chorus aus MIDI-Ereignissen erstellt, basierend auf den beiden Individuen. Außerdem liest GenJam ein progression file, in dem Tempo, Rhythmic Style (Swing oder gerade Achtel), Akkordfolgen und Tonart festgelegt sind. Zusätzlich kann es noch eine MIDI Sequenz für die Rhythmusgruppe einlesen.
Während man nun einem Solo zuhört, kann die Taste `g` (good) einmal oder öfter betätigt werden, wenn einem die momentan gespielte Phrase gefällt, oder die Taste `b` (bad) wenn nicht. Diese Fitnessdaten werden gespeichert und nach jedem Chorus ausgewertet.
Am Anfang dieses interaktiven Trainingsprozesses wird man nun fast alle viel versprechenden melodischen Ideen belohnen. Typischerweise taucht um die achte oder zehnte Generation eine „goldene“ Generation auf, in der fast alle Phrasen sehr musikalisch klingen. Ab diesem Zeitpunkt wird man nun nur mehr wirklich interessante Phrasen belohnen und auch vorher belohnte wieder bestrafen, wenn sie zu oft verwendet wurden.

ESSynth
(siehe http://www.nics.unicamp.br/papers_nics/waveform_synthesis.pdf)

ESSynth ist ein Wavetable-Synthese Instrument, das IGA verwendet.
Zuerst muss der Benutzer Initial Waveforms und Target Waveforms spezifizieren, nach denen sich die Sounds entwickeln sollen. Diese Target Waveforms sind beliebige, vom Benutzer festgelegte Soundfiles und können natürlich laufend geändert werden, sodass dauernd neue Klänge entstehen.
Der Computer produziert nun Generationen von Waveforms und verwendet dabei die Target Waveforms für die Bewertung der Fitness der einzelnen Individuen.
Es gibt nun drei elementare Kontrollstrukturen (siehe Skizze): die Population der n-ten Generation, bezeichnet als $\mathbf{B}_{(n)}$ (die Anfangsgeneration wird mit $\mathbf{B}_{(0)}$ bezeichnet), die Target Waveforms **T** und die *Fitness Function* $\boldsymbol{f}$.
Die Fitness Function wird nun verwendet, um jeweils die beste Waveform **w*** jeder Generation $\mathbf{B}_{(n)}$ zu bestimmen.
Waveform **w*** ist definiert als die mathematisch nächste Waveform $\mathbf{B}_{(n)}$ zu den Target Waveforms, ausgerechnet von einer Distanzfunktion (Hausdorff Distance).
In jeder Generation wird nun die beste Waveform **w*** in den Buffer gespeichert und als periodisch gelesener Wavetable abgespielt.

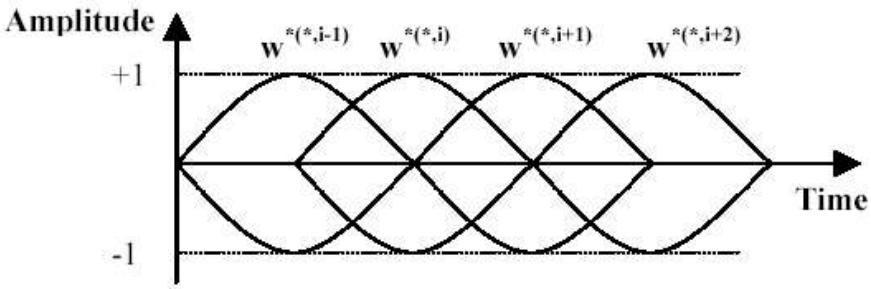
Ablaufskizze:[8]



Die Waveforms sind definiert als floating-point arrays mit 1024 points, im Intervall [-1, 1].
Die Initialpopulation $\mathbf{B}_{(0)}$ kann zufällig generiert oder benutzerspezifisch geladen werden.
Um nun die besten Waveforms jeder Generation $\mathbf{w}_{(*,i)}$ zusammenzufügen, wird die overlap-and-add Technik verwendet: ein Hanning-Fenster wird zu jeder Waveform $\mathbf{w}_{(*,i)}$ erstellt, sodass sich die Amplitude in den überlappenden Regionen nicht ändert (siehe Skizze).

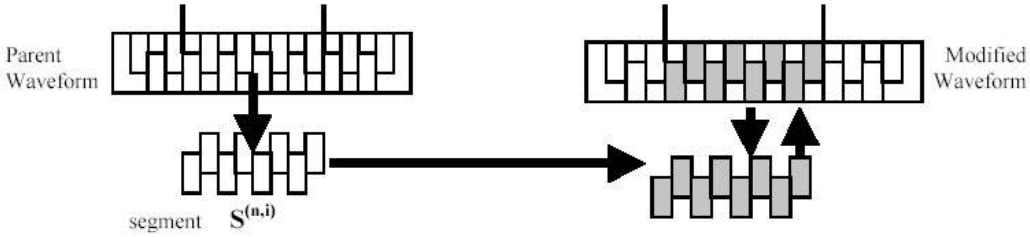
Overlap-and-add:[9]

[8] Bild von http://www.nics.unicamp.br/papers_nics/waveform_synthesis.pdf, Seite 2
[9] Bild von http://www.nics.unicamp.br/papers_nics/waveform_synthesis.pdf, Seite 2



ESSynth kreuzt die Population $\mathbf{B_{(n)}}$ mit den Elementen der Target Waveforms, um Ergebnisse zu bekommen, die **T** ähnlicher werden. Dabei wird ein Segment aus **T**, definiert zwischen den zufällig generierten Grenzen K1 und K2, mit dem equivalenten Segment von $\mathbf{B_{(n)}}$ vertauscht (Skizze).

Crossover:[10]



Um nun glitch noises in der neu entstandenen Waveform zu vermeiden, wird dem Segment $\mathbf{S_{(n,i)}}$ wieder ein Hanning-Fenster zugewiesen.

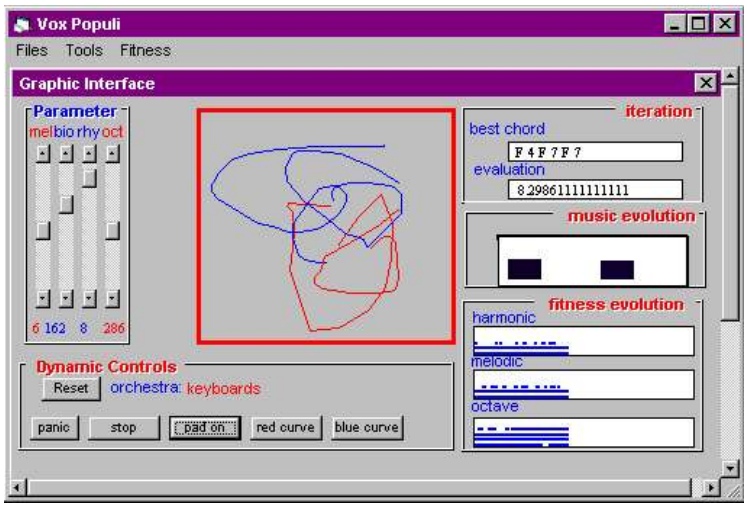
Vox Populi
(siehe http://gsd.ime.usp.br/sbcm/2000/papers/moroni.pdf)

In Vox Populi werden Populationen von Akkorden über MIDI generiert und entwickeln sich mit Hilfe der IGA in Real Time. Ein Fitnesskriterium ist definiert, das jeweils den besten Akkord einer Generation auswählt, dieser wird dann das nächste Element in der Sequenz, die gerade gespielt wird.
Eine Population ist aufgebaut als eine Gruppe aus vier Noten. GA werden verwendet, um eine Akkordfolge zu generieren und zu bewerten. Diese Akkordsequenz kann nun einer Kadenz oder einem Kontrapunkt ähneln. Vox Populi basiert auf einer Harmonik, in der eine melodische Linie als eine Annäherung zu einem tonalen Zentrum gedacht wird.
Im Selektionsprozess werden Gruppen von Stimmen mit der höchsten Fitness ausgewählt und abgespielt. Die musikalische Fitness eines Akkordes wird aus drei Teil-Fitnessfunktionen gebildet: Melodic-, Harmonic- und Voice Range Fitness.
Vox Populi besteht aus einem grafischen Interface, mit dem die Evolution der entstehenden Musik mit der Maus verändert werden kann (Abbildung).

Das Interface beinhaltet ein interaktives Pad, in dem zweidimensionale Kurven gezeichnet werden können. Die rote Kurve steuert die Melodic- und die Octave Range (siehe Slider mel, oct), die blaue Kurve Biological und Rhythmic Controls (siehe Slider bio, rhy)
Die Parameter können entweder über die Kurven des interaktiven Pads, oder „händisch“ mit den vier Slidern gesteuert werden.

[10] Bild von http://www.nics.unicamp.br/papers_nics/waveform_synthesis.pdf, Seite 3

Interface von Vox Populi:[11]



Ein genetic cycle besteht nun hautsächlich aus zwei Prozessen:

1) Entwicklung und Generierung der Individuen (Gruppen von Noten oder Stimmen) und Anwendung der genetischen Operationen an der Population
2) Das Interface, mit dem nun entschieden wird, welche Noten gespielt werden. Wenn eine Notengruppe ausgewählt wird, wird sie andauernd gespielt, bis die nächste Gruppe ausgewählt wird.

Das Timing dieser zwei Prozesse bestimmt nun hauptsächlich den Rhythmus der entstehenden Musik.
Vox Populi verwendet also den Computer und die Maus als real-time music controllers und kann als ein interaktives Computermusik Instrument gesehen werden. Der musikalische Output bewegt sich von sehr punktuellen Klängen bis hin zu breiten Akkorden, abhängig von der Anzahl der Individuen und der Dauer der einzelnen genetic cycles.

MutaSynth
(siehe http://www.design.chalmers.se/palle/publications/Dahlstedt-NowallsVisionPaper.pdf)

MutaSynth verwendet IGA zur Sound Synthese und Generierung von Pattern. Eine beliebige externe Sound Engine kann über MIDI mit MutaSynth mit Hilfe der genetischen Repräsentation ihrer Parameter gesteuert werden.

MutaSynth besteht also aus zwei Modulen:

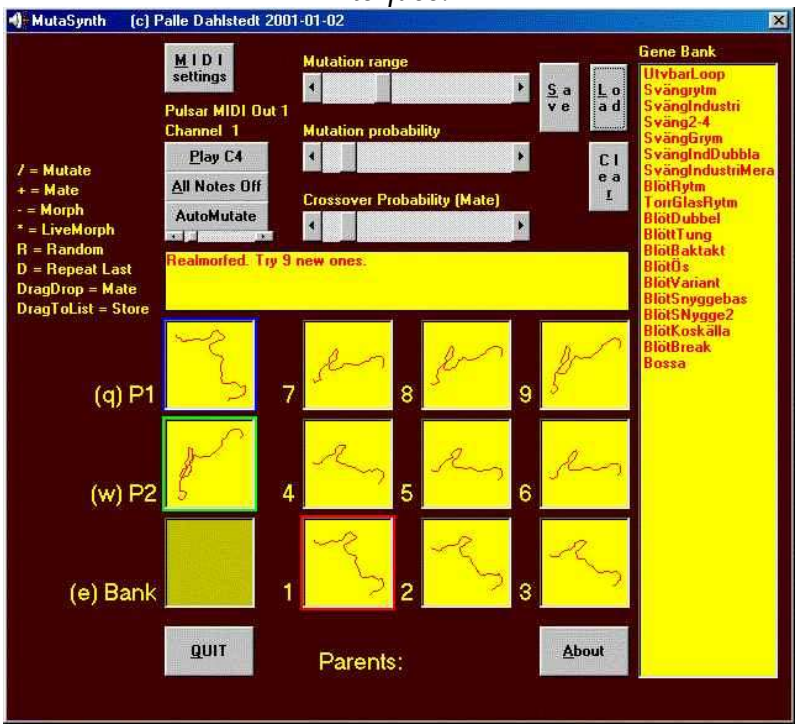
1) die (externe) Sound Engine
2) das Programm selbst, welches die genetischen Operationen durchführt

Eine Population in MutaSynth besteht aus neun Sounds oder Genomen, korrespondierend zu den neun Nummertasten am Keyboard. Durch Drücken der entsprechenden Taste kann der jeweilige Sound abgespielt werden.
Die genetischen Operationen in MutaSynth sind Mutation, Mating (Crossover), Insemitation (asymmetrical Crossover) und Morphing.
Mating und Insemitation sind zwei verschiedene Crossover Algorithmen. Bei Morphing wird eine lineare Interpolation jedes Gens der beiden Elterngenome durchgeführt und ein neues

[11] Bild von http://gsd.ime.usp.br/sbcm/2000/papers/moroni.pdf, Seite 5

Genom entsteht, indem an einer zufälligen Stelle im Parameterraum zwischen dem ersten und dem zweite Elternteil gestoppt wird.
Jeder genetische Operator ist manuell ansteuerbar und generiert eine neue Population an Genomen. Die Eltern, die von dem gewählten Operator benutzt werden, können zuvor gespeicherte Genome, ein Individuum in der aktuellen Population oder jede beliebige Einstellung der Sound Engine sein.

Interface:[12]



Das Interface zeigt neun Individuen, also die ganze Population, die zuletzt verwendeten Eltern und das momentan gewählte Genom in der Genbank.
Mit den Tasten +, -, * und / kann man die genetischen Operatoren starten.
Jedes Genom ist in einer wurmähnlichen Grafik dargestellt, um den Klang in einer gewissen Weise zu visualisieren und um somit die Auswahl zu erleichtern.
Um nun mit MutaSynth Kompositionen realisieren zu können, benötigt man eine externe Sound Engine, die mit MIDI gesteuert werden kann und eine geeignete Parameter zu Genom Transformation.

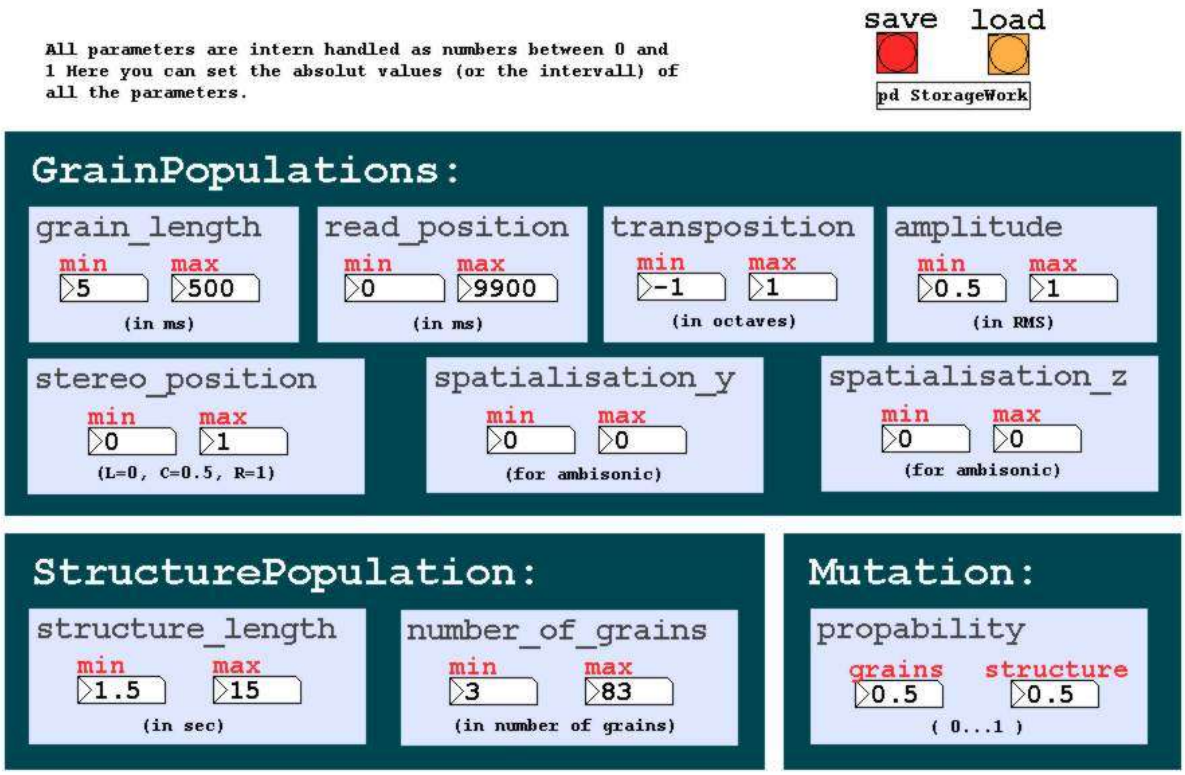
GenGran

GenGran ist eine „Improvisationsumgebung“ realisiert in PD[13] und verwendet zur Klangerzeugung Granularsynthese.
Grundsätzlich besteht GenGran aus parallelen Populationen: sechs Grain Populationen und einer Structure Population.
Die Structure Population triggert nun die einzelnen Grains, währenddessen die Grainpopulationen die Klangeigenschaften der Grains beinhaltet. Außerdem gibt es wiederum sechs parallele Grainpopulationen (entspricht sechs verschiedenen Stimmen), die auf

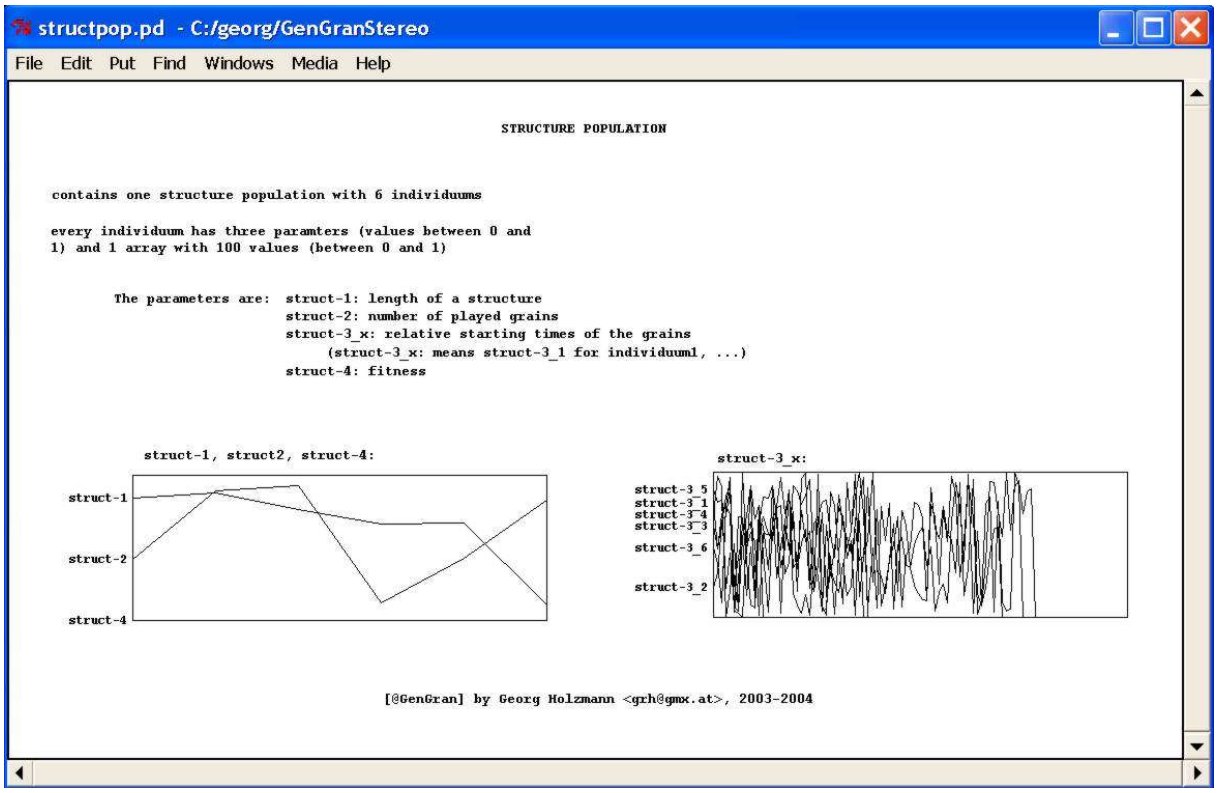
[12] Bild von http://www.design.chalmers.se/palle/publications/Dahlstedt-NowallsVisionPaper.pdf, Seite 4
[13] Pure Data (Miller Puckette), siehe http://www.pure-data.org

verschiedene Samples zugreifen und deren Parameter getrennt gesteuert werden können. Dadurch kann jeder Stimme eine charakteristische Klangfarbe zugeordnet werden.
Man „improvisiert“ nun mit GenGran, indem man im Interface verschiedene Stimmen auswählt, die Fitnessfunctions der Parameter zeichnet, die Populationen fortpflanzen lässt„… Außerdem ist es möglich bestimmte Konstellationen von Populationen abzuspeichern, um sie später wieder zu laden.

Eine Grainpopulation besteht nun aus einem Sample und folgenden Parametern: Grainlength (in ms), Readposition im Sample (in ms), Transposition, Amplitude, Spatialisation (ist ein Parameter in der Stereo-Version, bzw. drei in der Ambisonic-Version für R, Phi und Theta) und einem internen Fitness-Parameter, der die Fitness des Grains beinhaltet.
Grundsätzlich werden alle Parameter intern als Zahlen zwischen 0 und 1 verarbeitet, erst zum Schluss werden sie auf die eingegebenen „absoluten Parameter“ gemapt:



Weiters besteht jedes Structureindividuum aus den Parametern Structure length (in sec), Number of Grains, einem Array, das die Startzeiten der einzelnen Grains beinhaltet und natürlich dem Fitness-Parameter.



Diese Strukturen werden repetitiv verwendet, d.h. Structure length bestimmt die Länge der Repetition und Number of Grains die Anzahl der vorkommenden Grains darin. Die genaue Position dieser innerhalb der Struktur ist im Array struct-3_x (siehe Bild oben) gespeichert. Insgesamt gibt es in der Structurepopulation sechs Individuen, so kann jeder Stimme (d.h. jeder Grainpopulation) eine andere Struktur zugeordnet werden (willkürliche Zuordnung: Structureindividuum 1 zu Grainpopulation 1, Structureindividuum 2 zu Grainpopulation 2, …)

GenGran kann zufällig initialisiert werden (d.h. für jeden Parameter wird ein Zufallswert zwischen 0 und 1 bestimmt), oder es werden gespeicherte Populationen geladen.

Wenn eine Fortpflanzung ausgelöst wird, werden folgende 4 Schritte durchlaufen:

1) Fitness Scaling:

Von jedem Grain (bzw. Structure) in der jeweiligen Population wird die Fitness bestimmt. Dabei wird die gesamte Fitness aus dem Mittelwert der Fitness der einzelnen Parameter berechnet, welche durch die jeweilige gezeichnete Fitnessfunction im Interface berechnet wird.

2) Ordnen der Individuen:

In einem zusätzlichen Array wird nun die Fitnessreihenfolge der Grains (bzw. Structures) gespeichert, damit man dann sofort auf den Fittesten, Schlechtesten, … zugreifen kann.

3) Crossover:

Bei den Grainpopulationen werden dabei die fünf fittesten Paare ausgewählt (entspricht den zehn fittesten Grains) und ersetzen die fünf schlechtesten Paare. Die Structurepopulation erzeugt nur ein Nachkommen von dem fittesten Paar und ersetzt ein zufällig gewähltes Structureindividuum (hat sich musikalisch am sinnvollsten erwiesen).

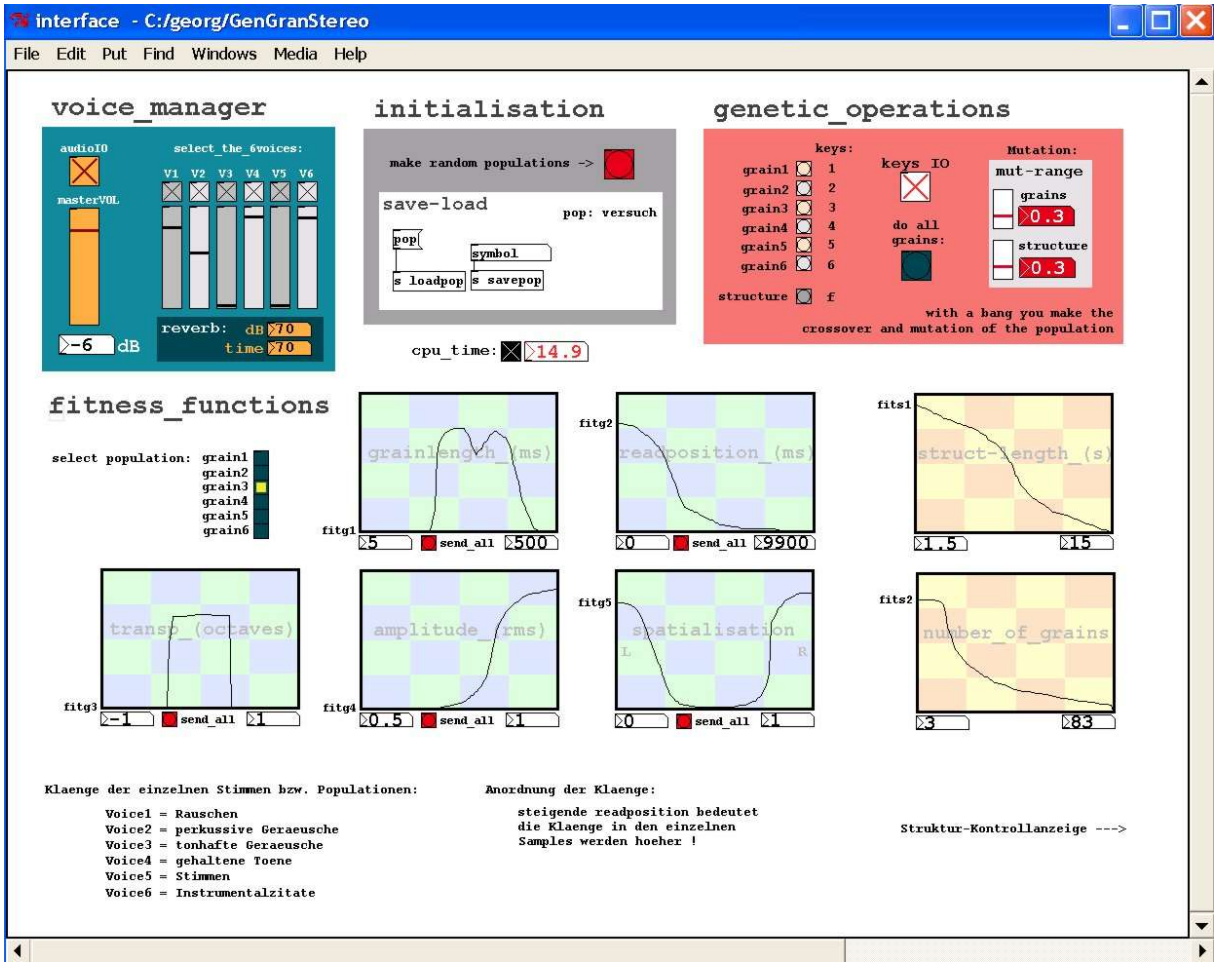
Das Crossover selbst funktioniert, indem mit den beiden Elternwerten des jeweiligen Parameters eine Zufallszahl bestimmt wird, die zwischen den Werten liegt. Bei den Grains entstehen zwei Nachkommen aus zwei Eltern, d.h. der zweite Wert wird ebenfalls zufällig aus diesen Schranken ermittelt.

4) Mutation:

Schlussendlich wird die Population noch mutiert. Diese Mutation wird von zwei Parametern gesteuert: Mutation-Probability (von 0 bis 1) und Mutation-Range (ebenfalls von 0 bis 1).
Mit der Mutation-Probability wird festgelegt, mit welcher Wahrscheinlichkeit der bestimmte Parameter mutiert wird.
Falls das der Fall ist, wird eine Zufallszahl innerhalb der Mutation-Range (alle Parameter werden ja als Zahlen zwischen 0 und 1 dargestellt) hinzu addiert oder subtrahiert.

Beschreibung des Interfaces:

Hier ist nun eine Abbildung des Interfaces für die Stereoversion von GenGran:



voice_manager:

Hier wird die Masterlautstärke, bzw. die Lautstärken der 6 Stimmen (entspricht den 6 Grainpopulationen) gesteuert. Zusätzlich kann man noch ein Reverb hinzufügen.

initialisation:

Mit „make random population“ werden alle Populationen zufällig initialisiert. Weiters besteht die Möglichkeit, vorher gespeicherte Populationen zu laden.

genetic_operations:

Mit den Buttons kann man die Fortpflanzung der einzelnen Populationen auslösen. Es besteht auch die Möglichkeit, die Keyboard Shortcuts (für Grains die Nummern „1, 2, 3, 4, 5, 6“ und für Structur „f“) zu verwenden, wenn der Toggle „keys IO“ aktiviert ist. Die Mutation-Range kann hier auch noch gesteuert werden (für alle Grainpopulationen zusammen und für die Structurepopulation). Damit kann man verhindern, dass der Algorithmus in lokalen Maxima hängen bleibt.

fitness_functions:

Hier können die Fitnessfunctions der einzelnen Parameter gezeichnet werden.
Die beiden rötlichen Arrays beinhalten die Fittnessfunctions für die Structurepopulation (struct-length, number_of_grains).
Die bläulichen Arrays für die Grainspopulationen, wobei die gewünschte Population (bzw. Stimme) zuerst mit dem Radiobutton „select population“ (links im Bild) ausgewählt werden muss.
Wenn man die gezeichnete Fitnessfunction eines bestimmten Parameters auf alle Grainpopulationen anwenden will, muss man den roten Button „send_all“ unter dem Array betätigen.

Literaturquellen

Genetic Algorithms, by John H. Holland
http://www.arch.columbia.edu/DDL/cad/A4513/S2001/r7

Genetische Algorithmen
http://www.chevreux.org/diplom/node32.html

GAlib: A C++ Library of Genetic Algorithm Components
http://lancet.mit.edu/ga/

Creating and Exploring the Huge Space Called Sound:
Interactive Evolution as a Composition Tool (Palle Dahlstedt)
http://www.design.chalmers.se/palle/publications/Dahlstedt-NowallsVisionPaper.pdf

Composing with Interactive Genetic Algorihms (Artemis Moroni, Jônatas Manzolli, Fernando Von Zuben)
http://gsd.ime.usp.br/sbcm/2000/papers/moroni.pdf

Sonomorphs: An Application of Genetic Algorithms to the Growth and Development of Musical Organisms (Gary Lee Nelson)
http://timara.con.oberlin.edu/~gnelson/papers/morph93/morph93.pdf

Waveform Synthesis using Evolutionary Computation (Jônatas Manzolli, Adolfo Maia Jr., Jose Fornari, Furio Damiani)
http://www.nics.unicamp.br/papers_nics/waveform_synthesis.pdf