1 Genetischer Algorithmus

Der genetische Algorithmus dient zur iterativen Optimierung der Testfälle. Er erzeugt eine Anzahl Chromosome, die über mehrere Stufen durch diverse Operationen verbessert werden.

1.1 Architektur

Der Genetische Algorithmus wird als eine Klasse implementiert (s. Abbildung 1).
In der Mutation- und ReproductionRate wird festgelegt, wie stark ein Chromosom mutieren soll oder wie viele neue Chromosome der Population pro Epoche hinzugefügt werden, bevor die Population wieder auf Soll-Größe schrumpft.
Da die Implementierung der einzelnen Schritte auf Modularität ausgelegt ist (s. Absatz 1.3), werden einige der Funktionen als Eigenschaften gesehen, da sie modular austauschbar sind.
Die Ausführung des Genetischen Algorithmus findet in der main Methode statt (s. Abbildung 2). Diese akzeptiert unter Umständen noch Parameter, welche zur Berechnung des Abbruchkriteriums genutzt werden. Aktuell bricht der genetische Algorithmus stets nach einer übergebenen Anzahl Epochen ab. Dies kann aber angepasst werden, falls es im weiteren Verlauf des Praktikums noch vonnöten wäre.

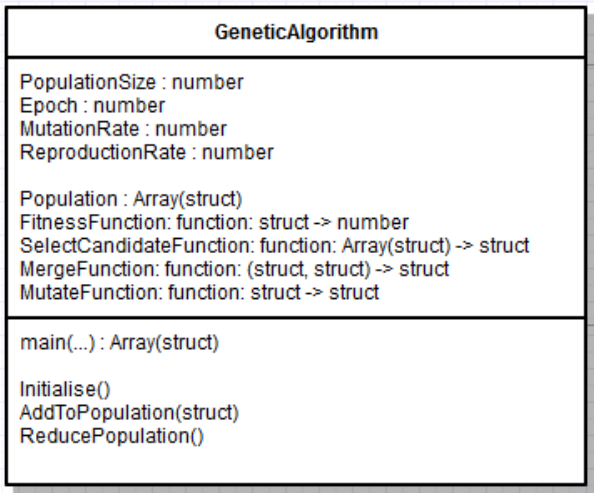

Abbildung 1: Klassenkarte des Genetischen Algorithmus

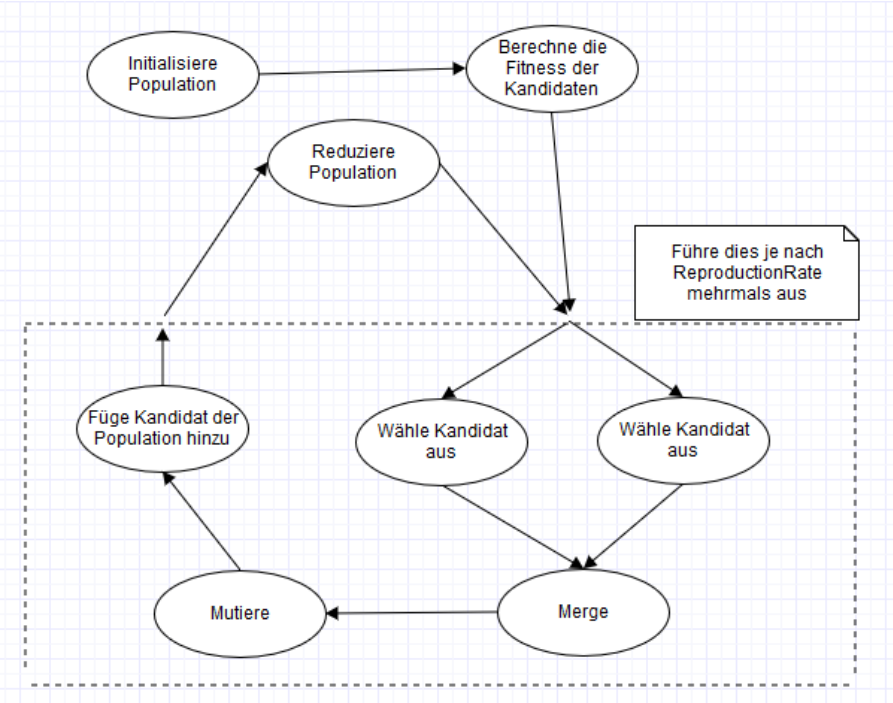

Abbildung 2: Ablauf des genetischen Algorithmus

1.2 Chromosome

Chromosome dienen der Persistierung der Testfall-Konfigurationen. Sie enthalten die Bestandteile eines realen Testfalls – so wie er für die Simulation benutzt wird – in einer für den Genetischen Algorithmus geeigneten Kodierung. Abbildung 3 bietet einen Überblick über die Bestandteile.

Chromosome
carx
cary
carangle
slotlength
slotdepth

Abbildung 3: Bestandteile eines Chromosoms

Die Information in den Chromosomen sind als prozentualer Anteil eines Wertebereichs kodiert. Die Unterteilung der Wertebereiche erfolgt durch eine 8-Bit Diskretisierung, somit entsprechen 0% dem Wert 0 und 100% dem Wert 255. Tabelle 1 stellt die tatsächliche Verteilung der Werte dar.

Wert	Minimum	Maximum
Fahrzeug X-Koordinate	-7.5	+7.5
Fahrzeug Y-Koordinate	-1	+4
Fahrzeug Orientierung	0	$2\,\pi$
Slot-Länge	2.25	5
Slot-Breite	1	2

Tabelle 1: Wertebereiche der Chromosom-Bestandteile

Die Chromosomen werden in Matlab in einer eigenen Klasse abgelegt. Sie speichert die zugewiesenen prozentualen Anteile an der Gesamt-Range sowie die zugeordnete Fitness. Zudem wird eine Funktion angeboten, die die Berechnung des durch das Chromosom repräsentierten Szenarios – und somit das Mapping auf die spezifizierten Wertebereiche – vornimmt.

1.3 Austauschbare Operatoren

Um eine skalierende Lösung vorzuhalten – insbesondere im Hinblick auf Erweiterungen und Optimierungen – wurden die Teil-Implementierungen des Evolutionsprozesses austauschbar realisiert. Sie werden als *Function-Handle* im Evolutionsframework verankert und können zu Beginn oder nach einer gewissen Epoche durch eine Funktion mit äquivalenter Signatur getauscht werden. So ist es Beispielsweise möglich, Anfangs nach vielversprechenden Bereichen zu suchen, um im späteren Verlauf eine lokale Optimierung durchzuführen.

Operatoren Erzeugung Die Operatoren werden durch das *Factory-Pattern* generiert. Dabei wird eine Fabrik-Funktion verwendet um die Operation gegebenenfalls zu konditionieren. Sie liefert als Resultat die fertige Operation, die in den genetischen Algorithmus eingebunden werden kann.

2 Operatoren

Die Operatoren, die im genetischen Algorithmus angewendet werden, sind modular tauschbar. Sie müssen lediglich die Schnittstellendefinition einhalten.

2.1 Initialisierung

Erklärung Die Initialisierung erzeugt die initiale Population aus N Chromosomen für den Start vom genetischen Algorithmus. Für die Erzeugung der Population werden N Chromosome mit zufälligen Werten der notwendigen Bestandteile (carx, cary, carangle, slotdepth, slotlength) erzeugt.

Implementierung Die Initialisierung wird in der Methode Init der Klasse GeneticAlgorithm implementiert. Die Eigenschaft Population wird als leeres Chromosomen-Array erstellt und in N Schleifendurchläufen befüllt. Je Iteration wird für jedes Bestandteil eines Chromosoms ein zufälliger Integerwert aus einer Uniformverteilung im Bereich [0, 255] generiert. Diese Werte werden im Folgenden an den Chromosomkonstruktor übergeben.

Probleme Die Erzeugung eines Chromosoms mit voneinander unabhängig generierten Werten für dessen Bestandteile kann dazu führen, dass manche Chromosome für die Simulation nicht brauchbar sind. Allerdings würden solche Chromosome im ersten Durchlauf nach Auswertung der Fitnessfunktion vom genetischen Algorithmus verworfen werden.

2.2 Selektion

Erklärung Die Selektion wählt die Chromosome aus der aktuellen Population aus, aus welchen man neue Chromosome generieren wird. Auch hier wird Modularität ermöglicht, obwohl es einen Standardansatz gibt. Die Schnittstelle hierfür ist folgendermaßen definiert:

Select: array(chromosom) $\rightarrow$ chromosome

1. Die effizienteste und in vielen Fällen sogar ausreichende Selektionsfunktion wählt mit gleichverteilter Wahrscheinlichkeit ein Chromosom x aus der Population *ppl* aus
$$p(x) \;=\; \frac{1}{|ppl|}$$
wobei $|ppl|$ die Menge an Elementen der Population ist.

2. Die dem Standard entsprechende Funktion wählt ein Chromosom x der Population *ppl* mit Wahrscheinlichkeit
$$p(x) \;=\; \frac{Fitness(x)}{\sum_{c \,\in\, ppl} Fitness(c)}$$

2.3 Rekombination

Erklärung Die Rekombination führt den Crossover Teil des Genetischen Algorithmus aus. Er erstellt ein neues Chromosom wobei er für jeden einzelnen Wert entscheidet, welchem der gegebenen zwei Chromosome er diesen Wert entnimmt.
Die Schnittstelle für die Rekombination ist folgendermaßen definiert:

merge: (chromosome, chromosome) $\rightarrow$ chromosome

Probleme Die Rekombination wählt jeden Wert unabhängig von der Wahl der anderen Werte aus. Daraus kann sich das Szenario ergeben, dass die X-Position des neuen Chromosoms nicht vom gleichen Chromosom kommt, wie die Y-Koordinate. Hieraus folgt dasselbe Problem wie bei der Mutation, dass semantisch ähnliche Variablen vollständig unabhängig voneinander gesetzt werden. Auch hierfür kann man bei Bedarf die Modularität des Systems ausnutzen und eine erweiterte Crossover Funktion erstellen. Zur Zeit erweist sich die hier gewählte Funkion aber als den Anforderungen genügend.

2.4 Mutation

Erklärung Die Mutation genügt der Signatur

mutate: chromosome → chromosome

Durch die generische Auslegung der Mutation im Rahmen des genetischen Algorithmus ist ein Austausch jederzeit möglich. Für den aktuellen Stand wurde ein Algorithmus entwickelt, der das Genom als Integer interpretiert und in dessen Binärdarstellung – als String – umwandelt. Dieser String wird iterativ verarbeitet. Dabei wird jedes Bit mit einer zuvor definierten Wahrscheinlichkeit umgekehrt.

Probleme Die Mutation erfolgt mit einer identischen Wahrscheinlichkeit für hoch- und niederwertige Bits. Dies kann von Vorteil sein, um randomisiert auch große Sprünge zu erlauben. Allerdings können somit auch vielversprechende Kandidaten stark verändert werden. Es kann also eventuell Notwendig sein, Spezialformen der Bit-Mutation zu integrieren. Das modulare System bietet hierfür hervorragende Skalierungsmöglichkeiten.

2.5 Fitness

Erklärung Die Fitness dient als Grundlage für den Selektionsprozess. Sie erstellt aus einem Chromosom eine Simulation und wertet diese entsprechend ihrer Implementierung aus.
Das Fitnessfunktion genügt der Signatur

Fitness: chromosome → number

Außerdem können beliebig viele Fitnessfunktionen kombiniert werden, indem man ihre einzelnen Ausgaben miteinander multipliziert.

1. In erster Instanz wurde die Fitnessfunktion als umgekehrt proportional zum minimalen Abstand realisiert.

   Implementierung

   $$Fitness = \frac{1}{minDist + maxValue^{-1}}$$

   Die Skalierung mit $maxValue^{-1}$ legt den Maximalwert im Kollisionsfall fest und definiert damit die obere Schranke für die Fitnessfunktion. Der minimale Abstand wird in der Simulation ermittelt.

Probleme Diese Wahl der Fitnessfunktion führt dazu, dass ein Testfall auch als besonders gut eingestuft wird, wenn er mit Heck direkt an einem Hindernis beginnt und danach eigentlich einen sehr simplen Einparkvorgang durchführt.
Außerdem werden hier alle Werte ≥ 0 als sehr gut eingestuft, solange sie nur möglichst klein sind. Das ist natürlich nicht realitätsgetreu, da Werte der Größenordnung 10^{-5} bereits durch minimale Störeinwirkungen wie zum Beispiel starker Wind bereits zu einem Unfall führen können.

2. Als Alternative wurde eine Fitnessfunktion definiert, die es ermöglicht einen bestimmten minimalen Abstand zu bevorzugen. Hierbei wurde darauf geachtet, dass für Werte kleiner als der gewünschte Abstand die Fitnessfunktion einen „schlechteren" Wert berechnet, als für den gewünschten Abstand und Abstände, welche nahezu gleich 0 sind, möglichst schlecht darzustellen. Für Abstände größer oder gleich dem gwünschntem Abstand wurde der gleiche Ansatz wie in Variante 1 gewählt.

Implementierung

$$Fitness = \begin{cases} \frac{minDist}{desiredDist}^4 & \text{für } minDist < desiredDist \\ \frac{desiredDist}{minDist} & \text{sonst} \end{cases} \tag{1}$$

Im ersten Fall wird mit dem Exponenten 4 bewirkt, dass die Bestrafung für Abweichungen von der erwünschten Distanz wesentlich härter bestraft werden, wenn sie Richtung 0 gehen, als wenn sie Richtung ∞ gehen. Dies bewirkt, dass man sich bei der Wahl der erwünschten minimalen Distanz wesentlich zielstrebiger und mutiger verhalten kann, da sie eher über- als unterschritten wird.

Probleme Auch wenn nun Werte der Größenordnung 10^{-5} nicht mehr als sehr gut angesehen werden (sofern man das nicht explizit erreichen will), können immer noch Testfälle generiert werden, die mit optimalem Abstand beginnen und danach trivial ablaufen.

3. Eine weitere Möglichkeit für die Fitnessfunktion ist gegeben durch die Funktion, die den Abstand des Fahrzeugs zur Parklücke zu Beginn des Testfalles bewertet. Hierbei gilt: Je näher das Auto der Parklücke, desto komplexer ist der Einparkvorgang, desto besser ist der Testfall. In der Praxis gilt dies zwar nur, wenn wirklich Manövrierbedarf besteht, da die Implementierung des Parkassistenten jedoch bekannt ist, ist bekannt, dass dieser auf jeden Fall minimal aus- und wieder einparken wird.

Implementierung Für ein Chromosom *chr* gilt:

$$Fitness = \frac{1}{|(x_{target}, y_{target}) \ - \ (x_{start}(chr), y_{start}(chr))} \tag{2}$$

Probleme Diese Implementierung führt zu Konvergenz in stets die gleiche Richtung. Die Testfälle werden sich nach ∞ Iterationen nicht mehr in der Position des Fahrzeuges zu Beginn unterscheiden. Außerdem berücksichtigt diese Fitnessfunktion nicht die Durchführung des Testfalles und kann - unter der Annahme, dass sinnvolle Testfälle gesucht werden - nicht alleine verwendet werden.

4. Eine derzeit letzte Implementierung der Fitnessfunktion bewertet die Größe der Parklücke. Sie gibt kleinen Parklücken bessere Werte als größeren. Dies macht intuitiv bereits den Einparkvorgang schwerer und reduziert auch die durchschnittliche Distanz des Fahrzeugs zu Hindernissen über die gesamte Dauer des Testfalles.

Implementierung Für ein Chromosom *chr* gilt:

$$Fitness = \frac{1}{Length_{slot}(chr) + Depth_{slot}(chr)} \tag{3}$$

Probleme Diese Implementierung führt dazu, dass Rückwärts-Einpark-Szenarien bevorzugt werden, da die Parklücke in diesem Fall kleiner sein kann. Außerdem gilt auch für diese Fitnessfunktion, dass sie nicht alleine verwendet werden kann, sofern man sinnvolle Ergebnisse berechnen will.