##### 

**Vorlage für die DIPLOMARBEIT**

**an der HTL Anichstraße**

**Version: 1.4**

**Gültig ab: 31.07.2021**

**Verteilermethode: schulintern – moodle**

**Dokumentenstatus: freigegeben**

**Es darf nur mit aktuellen Originaldokumenten gearbeitet werden.**

**Diese Seite und die folgenden 2 Seiten dienen der Dokumentenlenkung und müssen in der Diplomarbeit gelöscht werden.**

**Änderungshistorie:**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Datum** | **Änderungsgrund** | | **Änderungsstand** | | **Verantwortliche** |
| **28.11.18** | **Rechtschreib- und Formatierungsfehler** | | **V1.1** | | **YH** |
| **11.03.19** | **Inhaltsverzeichnis an V1.0 angepasst** | | **V1.2** | | **YH** |
| **21.03.20** | **Seite 4 angepasst Kopfzeile** | | **V1.3** | | **YH** |
| **15.07.21** | **Neues Logo** | | **V1.4** | | **YH** |
| Erstellt: | | | geprüft | | Freigegeben: | | |
| 15.07.21, YH | | | 15.07.21, YH | | 15.07.21, YH | | |
| Datum, Ersteller | | | Datum, Qualitätsbeauftragte | | Datum, Direktion | | |

Information zur nachfolgenden Dokumentation der Diplomarbeit

Sehr geehrte Diplomandinnen, sehr geehrte Diplomanten,

alle Abteilungen der HTL Anichstraße haben sich entschlossen, eine einheitliche Dokumentation der Diplomarbeiten einzuführen.

Dafür wurde nachfolgende Vorlage erarbeitet, die am Moodle Server zur Verfügung gestellt wird.

Die blau geschriebenen Texte sind zur Unterstützung gedacht und müssen in der Diplomarbeit auf die Standardtextfarbe (schwarz) umgestellt werden. Das Inhaltsverzeichnis ist bindend, kann aber natürlich mit Unterpunkten beliebig erweitert werden. Optionale Teile können entfallen.

Die Vorlage ist verpflichtend zu verwenden!

**Zusatzinformationen:**

**Wie zitiere ist?**

Es gibt unterschiedliche Vorgaben für ein Zitat, die HTL Anichstraße bezieht sich auf die Regeln der Diplomarbeitswebseite des Bildungsministeriums und diese Zitierregeln sind anzuwenden:

<http://www.diplomarbeiten-bbs.at/hinweise-zum-wissenschaftlichen-arbeiten/zitation-plagiate>

**FAQ**

Auf der vom Ministerium ausgearbeiteten Seite sind sehr viele Fragen zur und über die Diplomarbeit sehr gut beantwortet. Bei Unklarheiten wenden Sie sich aber gerne an die Betreuerin, den Betreuer oder Abteilungsvorstand.

<http://www.diplomarbeiten-bbs.at/faq>

**Gendern**

Aus Gründen der besseren Lesbarkeit wird in dieser Diplomarbeit die Sprachform des generischen Maskulinums angewendet. Es wird darauf hingewiesen, dass die ausschließliche Verwendung der männlichen Form geschlechtsunabhängig verstanden werden soll.

**Sperrvermerk**

Der Sperrvermerk wird dann ins Dokument eingebunden, wenn die Auftraggeberin, der Auftraggeber dies aus verständlichen Gründen vorgibt. WENN ein Sperrvermerk notwendig ist, dann werden die Seiten, die es betrifft in der Bibliotheksversion NICHT in die DA eingebunden (Vorgabe der HTL Anichstraße), die Seiten bleiben LEER, damit beugen wir vor, dass ein gesperrtes Exemplar versehentlich öffentlich ausgegeben werden kann. Die Betreuerin, der Betreuer benötigen für die Notenfindung ein vollständiges Dokument.

Wir möchten darauf hinweisen, dass die Präsentation trotz Sperrvermerk ÖFFENTLICH ist!

**DIPLOMARBEIT**

**TONE**

**Höhere Technische Bundeslehr- und Versuchsanstalt Anichstraße**

**\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

**Elektronik**

**Elektronik und Informatik**

|  |  |  |
| --- | --- | --- |
| Ausgeführt im Schuljahr 2021/22 von: |  | Betreuer/Betreuerin: |
| Crispin Holleis  Manolo Jaschke  Nils Kuhn |  | Dipl.-Ing. Markus Signitzer |

Projektpartner: -

Innsbruck, am TT.MM.JJJJ

Abgabevermerk: Betreuer/in:

Datum:

**SPERRVERMERK**

Auf Wunsch der Firma Vollständige Bezeichnung der Firma ist die vorliegende Diplomarbeit

für die Dauer von drei / fünf / sieben Jahren

für die öffentliche Nutzung zu sperren.

Veröffentlichung, Vervielfältigung und Einsichtnahme sind ohne ausdrückliche Genehmigung der Firma \*\*\* und der Verfasser

bis zum TT.MM.JJJJ nicht gestattet.

Innsbruck, TT.MM.JJJJ

Verfasser:

Vor- und Zuname Unterschrift

Vor- und Zuname Unterschrift

Firma:

Firmenstempel

## Kurzfassung /Abstract

## Projektergebnis

**Inhaltsverzeichnis**

[Kurzfassung /Abstract i](#_Toc98691955)

[Projektergebnis i](#_Toc98691956)

[Einleitung 1](#_Toc98691957)

[Vertiefende Aufgabenstellung 1](#_Toc98691958)

[1.1 Crispin Holleis 1](#_Toc98691959)

[1.2 Manolo Jaschke 1](#_Toc98691960)

[1.3 Nils Kuhn 1](#_Toc98691961)

[2 NeuroEvolution 2](#_Toc98691962)

[2.1 Einführung 2](#_Toc98691963)

[2.2 Auswahl des Algorithmus 2](#_Toc98691964)

[2.3 NEAT 3](#_Toc98691965)

[2.3.1 Allgemeines 3](#_Toc98691966)

[2.3.2 Funktionsweise 3](#_Toc98691967)

[2.3.3 Genetic Encoding 4](#_Toc98691968)

[2.3.4 Speciation 5](#_Toc98691969)

[2.3.5 Minimal Structure 6](#_Toc98691970)

[2.4 Implementation 6](#_Toc98691971)

[2.4.1 Einleitung 6](#_Toc98691972)

[2.4.2 Programmarchitektur 7](#_Toc98691973)

[2.4.3 API 9](#_Toc98691974)

[2.4.4 Details 10](#_Toc98691975)

[2.4.5 Probleme 10](#_Toc98691976)

[2.5 Layout des neuronalen Netzwerks 10](#_Toc98691977)

[2.5.1 Komplexität 10](#_Toc98691978)

[2.6 Ergebnis 10](#_Toc98691979)

[2.6.1 Settings 10](#_Toc98691980)

[3 Erklärung der Eigenständigkeit der Arbeit 11](#_Toc98691981)

[I. Abbildungsverzeichnis I](#_Toc98691982)

[II. Tabellenverzeichnis I](#_Toc98691983)

[III. Literaturverzeichnis I](#_Toc98691984)

[IV. Abkürzungs- und Symbolverzeichnis I](#_Toc98691985)

[Anhang II](#_Toc98691986)

[A1 Pflichtenheft (OPTIONAL) II](#_Toc98691987)

[A2 Schlussfolgerung / Projekterfahrung II](#_Toc98691988)

[A3 Projektterminplanung II](#_Toc98691989)

[A4 Arbeitsnachweis Diplomarbeit II](#_Toc98691990)

[A5 Datenblätter (OPTIONAL) II](#_Toc98691991)

[A6 Technische Zeichnungen (OPTIONAL) II](#_Toc98691992)

Einleitung

Vertiefende Aufgabenstellung

## Crispin Holleis

Konzeptionierung und Implementieren des steuernden neuronalen Netzwerks, einschließlich kurzer Erläuterung der allgemeinen Funktionsweisen verwendeten Algorithmen im Rahmen der Dokumentation.

## Manolo Jaschke

Simulation des Verkehrs als Produkt individueller Ziele einzelner Fahrzeuge, außerdem Entwicklung einer koordinierenden Schnittstelle zwischen Simulation, neuronalem Netzwerk sowie Userinterface.

## Nils Kuhn

Konzeptausarbeitung eines Userinterfaces und Entwicklung der grafischen Oberfläche sowie Visualisierung des Simulationsvorganges und zugehöriger Daten mittels einer Engine.

# NeuroEvolution

Phäääänotyp; output zwischen 0 und 1

## Einführung

Der Straßenverkehr wird von einem mittels *Machine Learning*entwickelten Algorithmus koordiniert, dieser übernimmt die Aufgabe der Routenfindung zwischen einem beliebigen Start-Ziel-Punktpaar. Die entwickelte KI ist in der Lage, Fahrzeugen in einem zufällig generierten Straßennetz entsprechende Routen zu ihren jeweiligen Zielpunkten vorzugeben.

Das Straßennetz liegt in Rasterform mit festgelegter maximalen Größe vor.

## Auswahl des Algorithmus

Mit zunehmender Popularität von Machine Learning haben sich eine Vielzahl verschiedener Techniken entwickelt. Machine Learning ist ein Teilgebiet der künstlichen Intelligenz und bezeichnet Algorithmen, die sich durch Erfahrung und/oder Datenanalyse automatisiert verbessern können. Eine Verbesserung bedeutet in diesem Kontext einen Fortschritt beim Erreichen eines expliziten Ziels.[[1]](#footnote-1)

Die Aufgabe der Verkehrsoptimierung impliziert eine Restriktion, welche die möglichen Algorithmen stark einschränkt: Die Lösung ist nicht bekannt, nur die Qualität der Antwort kann gemessen werden. Dies entspricht einem *Reinforcement Learning* Szenario. Dieser Bereich des Machine Learning beschäftigt sich mit der Aufgabe, Entscheidungen zu treffen, welche zu einer maximalen Belohnung führen. Zur Berechnung der Belohnung dient oftmals eine *Fitness-Function*, welche den Erfolg einer gewissen Entscheidungskette quantifiziert. Der Lernprozess besteht aus der Optimierung verschiedener Parameter, mit deren Hilfe eine Lösung ermittelt wird.[[2]](#footnote-2)

Reinforcement Learning ist unter anderem mittels der biologisch-evolutionär inspirierten *Evolutionären Algorithmen (engl. Genetic Algorithms)* möglich*,* welche mittels Selektion und Operatoren wie *Mutation* und *Paarung* eine Lösung entwickeln. Eine gefundene Lösung ist hierbei nicht garantiert ideal, für die spezifische Aufgabe jedoch meistens ausreichen.

Der große Vorteil von genetischen Algorithmen gegenüber anderen Techniken liegt in der relativen Einfachheit ihrer Kernmechanismen. Da diese selbstständig implementiert werden sollen, wurde mit dem NEAT-Algorithmus ein Vertreter dieser Kategorie gewählt. Im folgenden Kapitel wird die Funktionsweise von NEAT zusammengefasst.

## NEAT

Das gesamte folgende Kapitel ist eine Zusammenfassung von Kenneth O. Stanley 2004, *Efficient Evolution of Neural Networks through Complexification*.

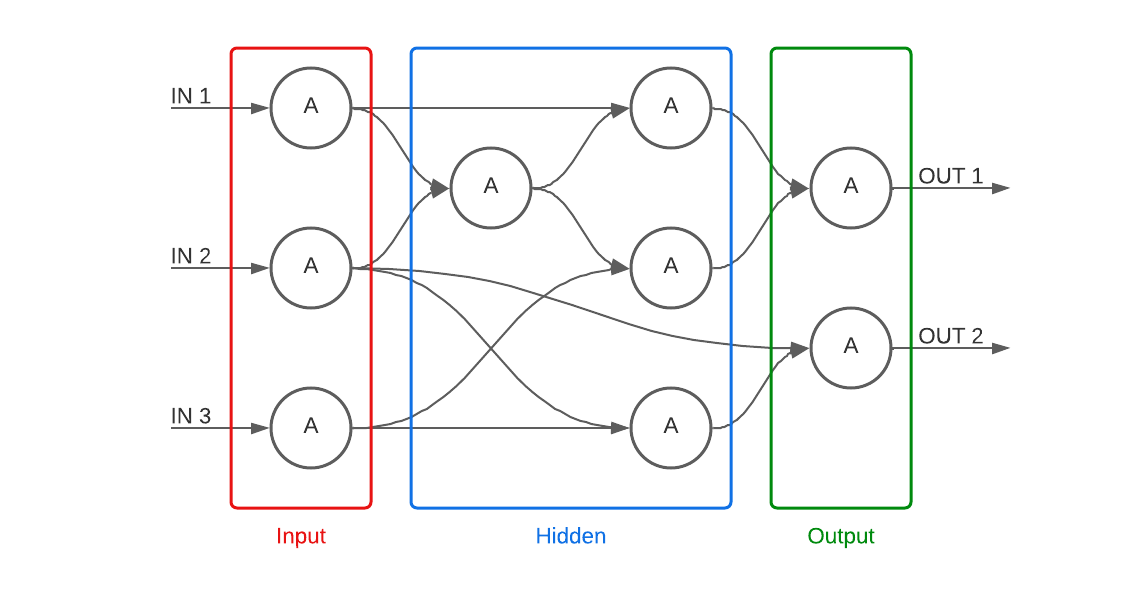
Seitenzahlen

### Allgemeines

Das Akronym NEAT steht für *NeuroEvolution of Augmenting Topologies* und ist ein genetischer Algorithmus zur Entwicklung künstlicher neuronaler Netzwerke.

Ein *neuronales Netzwerk* stellt eine Struktur mit einer fixen Anzahl von Ein- und Ausgängen dar, welches aus miteinander verbundenen *Neuronen* besteht (S. 9). Dieses Netz besteht aus zwei fundamentalen Komponenten:

* Die **Neuronen** fungieren als Knoten mit beliebig vielen Eingängen und einem Ausgang. Ist das Netzwerk aktiv, summieren die Neuronen ihre Eingänge auf und übergeben die Summe an eine nichtlineare *Aktivierungsfunktion*. Das Ergebnis liegt dann am Ausgang an. Neuronen, welche weder als Ein- noch als Ausgang fungieren, werden als *Hidden-Neurons* bezeichnet (S. 9).
* **Verbindungen** vernetzen ein Source- und ein Target-Neuron und verfügen über einen konstanten Faktor, ein sogenanntes *Weight*. Ihre Aufgabe besteht aus der Weiterleitung des Ausgangswertes des Source-Neurons an einen der Eingänge des Target-Neurons. Der weitergeleitete Wert wird hierbei mit dem Weight multipliziert. Je nach Vorzeichen kann die Verbindung folglich stimulierend oder hemmend wirken.



Im Fall von NEAT existiert zusätzlich ein *Bias*-Eingang, an welchem ein konstanter Wert anliegt.

Ein neuronales Netz von ausreichender Größe kann bewiesenermaßen jede kontinuierliche mathematische Funktion annähern (S. 8).

### Funktionsweise

Im Gegensatz zu vielen anderen neuroevolutionären Methoden optimiert NEAT nicht ein statisches neuronales Netzwerk, sondern erlaubt das Hinzufügen (und in einigen Implementierungen Löschen) von Neuronen und Verbindungen. Dies ermöglicht zusätzlich zur Anpassung der Gewichtungen die Entwicklung einer möglichst effizienten Topologie, führt allerdings auch zu einer Reihe von Schwierigkeiten (S. 8, 17-21, 24, 34).

Die Evolution geschieht in *Generationen,* welche eine gewisse Anzahl von Individuen enthalten, entsprechend der *Populationsgröße.* In jeder Generation wird die Fitness der Individuen bestimmt, je erfolgreicher diese sind, desto höher die Wahrscheinlichkeit auf Fortpflanzung in die nächste Generation (S. 7).

### Genetic Encoding

Genetische Kodierung bezeichnet die Art, auf die das neuronale Netzwerk gespeichert wird. Im Fall von NEAT muss diese Struktur zwei fundamentale Operatoren erlauben: *Mutation* und *Paarung (engl. mating* und *crossover)* (S 7, 11)*.*

* **Mutation** bezeichnet eine zufällige Veränderung eines Netzes. Dies kann die Veränderung eines Connection-Weights, das Hinzufügen (bzw. Entfernen) einer Verbindung sowie das Hinzufügen eines Neurons durch Teilen einer vorhandenen Verbindung sein.
* **Paarung** entspricht der Kreuzung zweier unabhängiger Netzwerke. Hierbei muss sichergestellt werden, dass das Ergebnis mit ausreichender Wahrscheinlichkeit aussagekräftig und sinnvoll ist.

Aufgrund der variablen Netzstruktur werden nicht nur die verschiedenen Verbindungsgewichtungen gespeichert, sondern auch die Topologie. Als Folge hiervon treten drei große Probleme auf:

* **Competing Conventions** bedeutet, dass unterschiedliche Layouts dieselbe Funktion erfüllen können. Dies ist beispielsweise bei gespiegelten Netzen der Fall (S. 18-20).
* Die **variable Länge** der entwickelten Lösungen bedeutet, dass auch die interne Repräsentation entsprechend dynamisch sein sollte, um einerseits keine willkürlichen Limitationen zu setzen und andererseits auch bei kleinen Topologien Ressourcen effizient nutzen zu können. Die Konsequenz dieser Anforderung ist die Unvereinbarkeit mit einigen klassischen Ansätzen wie beispielsweise Connection-Bitmasks (S. 17, 34).
* Zusätzlich ist beim **Crossover** sicherzustellen, dass der erzeugte Nachwuchs mit angemessener Wahrscheinlichkeit funktional ist. Hierzu ist eine Identifikation von funktionsgleichen Komponenten notwendig, um diese sinnvoll kreuzen zu können. Das Problem der *Competing Conventions* erschwert diese Aufgabe zusätzlich, denn auch Bestandteile mit derselben Funktion können unterschiedliche Layouts aufweisen.

NEAT löst diese Schwierigkeiten mit *Genomen* und dem Tracking ihrer Abstammung (S. 36-38).

Genome repräsentieren Verbindungen oder Neuronen des Netzwerks, stellen also eine Kodierung eines Multigraphs dar. Verbindungsgenome bestehen aus Source- sowie Target-Neuron, Weight und einem Enable-Bit, Neuronen-Genome lediglich aus einer Neuronen-ID.

Grafik Repräsentation

Die Ursprungsverfolgung erfolgt über die sogenannte *Innovation Number*, eine einzigartige Zahl, welche einer Verbindung beim Entstehen zugeordnet wird. Werden die Verbindungen vererbt, können sie mittels dieser Nummer auch nach beliebig vielen Generationen noch verglichen werden. Besitzen zwei Verbindungen dieselbe Nummer, so geht der Algorithmus auch von einer gleichen Funktion aus. Um Netzwerke auf diese Weise kompatibel zu halten, darf es auch bei den Neuronen-IDs keine Duplikate geben.

Der Crossover-Mechanismus nimmt das erfolgreichere Elternnetz (höhere Fitness) als Grundlage und bildet den arithmetischen Mittelwert der Gewichtungen aller übereinstimmenden Verbindungen. Bei gleicher Fitness werden alle nicht übereinstimmenden Genome mit 50 prozentiger Wahrscheinlichkeit übernommen (S. 34 – 35).

Grafik Mating

### Speciation

Ein weiteres Problem ist die Tatsache, dass neue Mutationen zu Beginn oftmals schädlich für das Fitness-Rating eines Netzwerks sind, auch wenn sie auf längere Sicht vorteilhaft wären. Dies ist der Fall, da hinzugefügte Verbindungen erst angepasst werden müssen, bevor sie ihre Funktion erfüllen können. Um dies zu ermöglichen, wird die Bevölkerung in Spezies unterteilt. Jede Spezies erhält einen gewissen Anteil des Nachwuchses, wobei größere Spezies proportional weniger Nachwuchs erhalten. Die Fitness einer Spezies errechnet sich als

wobei N die Anzahl der Individuen einer Spezies, fi die Fitness eines Individuums und fk die Fitness der gesamten Spezies darstellt. Im weiteren Verlauf sowie im Source-Code wird auf diese Formel als *Quadratdurschnitt (engl. square average)* verwiesen.

Grafik Fitnessverlauf

Der Nachwuchs einer Spezies wird als direkter Anteil an der Gesamtfitness berechnet:

Nk steht hierbei für die Anzahl an Spezies, fk für den entsprechenden Quadratdurchschnitt, P für die Populationsgröße und nk für die zugeordnete Nachwuchsanzahl.

Die Einteilung in Spezies geschieht mittels einer *Distance-Function* sowie einem *Compatibility-Threshold*. Unterschreitet der berechnete Unterschied den Schwellenwert, so sind gehören zwei Individuen zur selben Spezies. Der Unterschied ist abhängig von der Anzahl an unterschiedlichen Verbindungen (U) sowie der durchschnittlichen Weight-Differenz übereinstimmender Verbindungen:

Die originale NEAT-Implementation unterscheidet zusätzlich zwischen zwei unterschiedlichen Arten von differenten Verbindungen, verwendet jedoch denselben Zahlenwert als Koeffizienten, folglich kann dies zu obiger Formel vereinfacht werden (S. 38-40).

### Minimal Structure

Um eine möglichst effiziente Lösung, d.h. ein Netzwerk mit einer möglichst geringen Anzahl an Neuronen und Verbindungen zu finden, wählt NEAT den Ansatz einer minimalen Anfangsstruktur. Die Anfangspopulation besteht hierbei aus Netzwerken mit lediglich einer einzigen, zufällig erzeugten Verbindung. Dies vermeidet unnötige Komplexität, da die Topologie erst im Verlauf der Evolution graduell erhöht wird, solange kein hinreichendes Ergebnis erzielt wird. Zusätzlich verringert diese Technik den *Search Space,* da zu Beginn nur wenig veränderliche Parameter existieren und diese in späteren Nachkommen bereits angepasst sind (40 – 41).

## Implementation

### Einleitung

Die Grundfunktionalität des NEAT-Algorithmus steht wie im vorigen Kapitel erläutert bereits fest, dennoch gibt es einige Voraussetzung, die für einige Entscheidungen von Bedeutung sind.

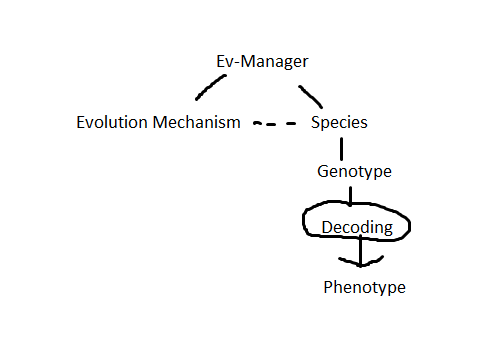
Der wichtigste Punkt ist die Performance. Für die Aufgabe der Verkehrsoptimierung ist ein verhältnismäßig großes neuronales Netzwerk zu erwarten, die Implementation muss fähig sein, dies zu bewältigen. Zusätzlich könnte Multithreading aufgrund des spezifischen Designs der Straßennetzsimulation nicht möglich sein, dies wäre ebenfalls durch ein leistungsfähigeres Programm auszugleichen.

Eine weitere essentielle Anforderung ist die Kompatibilität mit der Programmiersprache C#, da die graphische Oberfläche in der Grafik-Engine Unity und somit in C# erstellt wird. Das Programm muss also folglich entweder über ein entsprechendes Interface verfügen oder nativ kompatibel sein.

Zudem ist ein objektorientiertes Design anzustreben, da dies die Implementierung der komplexen Struktur vereinfacht und die originale Publikation bereits implizit eine Struktur umreißt. Außerdem ist C# rein objektorientiert, eine ähnliche Programmarchitektur vereinfacht die Implementation eines Interfaces.

Obwohl der NEAT-Algorithmus sich prinzipiell in jeder modernen Programmiersprache realisieren lässt, gibt es angesichts oben genannter Gründe bessere und weniger gut geeignete Kandidaten. Letztendlich fiel die Wahl auf C++, eine sehr performante, objektorientierte Sprache, welche mittels PInvoke (Platform Invoke) eine Interfacemöglichkeit zu C# besitzt.

### Programmarchitektur



Fehlt: Genome/Connection (?), Genotype Container

Diese Kernfunktionalität dieser Implementation ist in folgende Hauptkomponenten gegliedert:

* Der **Evolution Manager** koordiniert den **Evolution Mechanism** und fungiert als Container für die Individuen und Spezies. Außerdem stellt er das Lade- und Speicherfeature zur Verfügung.
* Die **Species**-Klasse ist für sämtliche Speciation-Funktionalitäten zuständig. Dies umfasst Distance-Berechnungen, Verwaltung von zugehörigen Individuen und die Berechnung des Quadratdurchschnitts.
* **IGenotype** definiert das Genotype-Interface. Ein Genotyp entspricht einer Sammlung von Genomen (siehe Kapitel) und ist somit die Repräsentation eines neuronalen Netzwerks. Die Hauptanforderungen sind Speicherung und Fortpflanzung, folglich fällt auch die Optimierung dementsprechend aus.  
  Zusätzlich verfügt die Implementation über einen **Genotype Container**, in welchem mehrere gleichartige Genotyp-Objekte gespeichert werden können.
* **Connection** ist ein simpler Datentyp zur Speicherung von Verbindungsinformationen. Er enthält lediglich Weight, Source-Neuron und Target-Neuron.
* Das **Genome** fungiert als Manager einer Connection. Genome sind für die Mutation von Verbindungen zuständig.
* **IPhenotype** ist das Interface des neuronalen Netzwerks. Phänotypen sind die funktionelle Entsprechung eines Genotyps. Im Gegensatz zu Genotypen sind sie für die Aktivierung, also den eigentlichen Betrieb optimiert.
* Unter **Decoding** fällt die Übersetzung eines Genotyps in einen Phänotyp. Um dies für jedwede Kombination zu ermöglichen, ist die Hilfsklasse **Generic Genotype** vorgesehen, welche von jedem Genotyp erstellbar und in jeden Phänotyp überführbar sein muss. Voraussetzung hierfür ist die Fähigkeit, sämtliche für die Übersetzung relevanten Informationen zu kodieren.

Wie bereits erwähnt ist eine ähnliche Gruppierung der Kernkomponenten auch in der ursprünglichen NEAT-Publikation vorhanden. Zusätzlich Inspiration, besonders hinsichtlich des Genotype-Phenotype-Dualismus stammt von der *SharpNEAT* Library von Colin Green.

### API

Dieses Kapitel ist eine Übersicht über die relevantesten Funktionen zur Verwendung als Library, das Grundkonzept findet sich in Kapitel 2.4.2. Eine detailliertere Beschreibung ist in Kapitel 2.4.4 zu finden. Aus Gründen der Übersichtlichkeit sind lediglich die relevanten C#-Funktionen dokumentiert.

Übersicht (Inhaltsverzeichnis?)

#### **NeatEvolutionManager:**

Der Evolution Manager ist C# als Klasse realisiert, folglich sind sämtliche Funktionen Klassenmethoden.

**Konstruktor:**

|  |  |
| --- | --- |
| **Signatur** | NeatEvolutionManager(uint t\_input\_num, uint t\_output\_num, NeatEvolutionSettings t\_settings) |
| **Funktion** | Erstellt eine neue Evolution-Manager Instanz. |
| **Verwendung** | Das Erstellen eines Evolution-Managers stellt den Start des Evolutionsvorganges dar. |
| **Argumente** | *t\_input\_num:* Die Anzahl an Eingängen des zu entwickelnde Netzwerk. Diese ist unveränderlich und muss größer als 0 sein.  *t\_output\_num:* Die Anzahl an Ausgängen des zu entwickelnde Netzwerk. Diese ist unveränderlich und muss größer als 0 sein.  *t\_settings:* Ein Objekt vom Typ NeatEvolutionSettings. Legt weitere Randparameter der Evolution fest. |
| **Returns** | - |
| **Threadsafe** | Nein |
| **Seiteneffekte** | Pro Programm sollte nur ein einziges Objekt vom Typ NeatEvolutionManager erstellt werden. Missachtung könnte zu verminderter Performance oder undefiniertem Verhalten führen. |

**Erzeugung einer zufälligen Population:**

|  |  |
| --- | --- |
| **Signatur** | void CreateRandomPopulation() |
| **Funktion** | Ersetzt die gegenwärtigen Individuen durch eine neue, zufällig generierte Population. |
| **Verwendung** | Erzeugt die Anfangspopulation bzw. entspricht einem Reset des Evolutionsvorganges. |
| **Argumente** | - |
| **Returns** | - |
| **Threadsafe** | Nein |
| **Seiteneffekte** | Invalidiert sämtliche Instanzen von SimplisticGenotype. |

**Zugriff auf die aktuelle Population:**

|  |  |
| --- | --- |
| **Signatur** | SimplisticGenotype[] GetPopulation() |
| **Funktion** | Liefert ein Array aller Genotypen der aktuellen Generation zurück. |
| **Verwendung** | Gewährt Zugriff auf die Genotypen einer Generation, um diese testen und bewerten zu können. |
| **Argumente** | - |
| **Returns** | Ein Array vom Typ SimplisticGenotype mit sämtlichen Individuen der aktuellen Generation als Inhalt. |
| **Threadsafe** | Nein |
| **Seiteneffekte** | - |

**Zugriff auf den fähigsten Genotyp:**

|  |  |
| --- | --- |
| **Signatur** | SimplisticGenotype GetPreviousBestGenotype() |
| **Funktion** | Liefert den Genotyp mit der höchsten Fitness der vorigen Generation zurück. Dieser wird mit jedem Aufruf von EvolvePopulation() neu ermittelt. |
| **Verwendung** | Kann beispielsweise für einen Vergleich mit den aktuellen Individuen genutzt werden. |
| **Argumente** | - |
| **Returns** | Ein Objekt vom Typ SimplisticGenotype. |
| **Threadsafe** | Nein |
| **Seiteneffekte** | - |

**Verarbeiten einer Generation:**

|  |  |
| --- | --- |
| **Signatur** | void EvolvePopulation() |
| **Funktion** | Ersetzt die aktuelle Population durch neue, gemäß dem Fortpflanzungsalgorithmus erzeugte Individuen. Zuvor muss sämtlichen Genotypen eine Fitness zugewiesen werden, um eine repräsentative Nachwuchsgenerierung zu ermöglichen. |
| **Verwendung** | Dies ist die Kernfunktion, welche eine neue Generation von Genotypen hervorbringt. |
| **Argumente** | - |
| **Returns** | - |
| **Threadsafe** | Nein |
| **Seiteneffekte** | Invalidiert sämtliche Instanzen von SimplisticGenotype. |

**Speichern:**

|  |  |
| --- | --- |
| **Signatur** | void Save(string t\_file) |
| **Funktion** | Speichert die aktuelle Population als XML-File. |
| **Verwendung** | Speichert den Evolutionsfortschritt, ein Fortsetzen des Vorganges ist später mittels Load() jederzeit möglich. |
| **Argumente** | *t\_file:* Ein String, welcher Filename und optional Path spezifiziert. Der Pfad kann sowohl relativ als auch absolut sein. Eine Dateiendung ist nicht zwingend notwendig, *.xml* wird aber empfohlen.­ |
| **Returns** | - |
| **Threadsafe** | Nein |
| **Seiteneffekte** | Überschreibt das gegebene File ohne Rückfrage, falls es bereits existiert. |

**Laden:**

|  |  |
| --- | --- |
| **Signatur** | void Load(string t\_file) |
| **Funktion** | Lädt die Individuen eines XML-Files als Population.  Hierbei ist darauf zu achten, dass ein File nur mit denselben Einstellungen geladen werden kann, mit denen es ursprünglich gespeichert wurde.  Hierbei ist darauf zu achten, dass die Rahmeneinstellungen des ladenden Evolution-Manager dieselben sind, mit denen das File ursprünglich gespeichert wurde.  Konkret müssen Bevölkerungsgröße sowie Anzahl an Ein- und Ausgängen übereinstimmen. Dies wird beim Laden nicht überprüft. |
| **Verwendung** | Lädt einen gespeicherten Evolutionsfortschritt und ermöglicht ein Fortsetzen vom letzten Stand aus. |
| **Argumente** | *t\_file:* Ein String, welcher Filename und optional Path spezifiziert. Der Pfad kann sowohl relativ als auch absolut sein. Die Dateiendung muss im Namen enthalten sein. |
| **Returns** | - |
| **Threadsafe** | Nein |
| **Seiteneffekte** | Überschreibt die aktuellen Individuen und invalidiert sämtliche Instanzen von SimplisticGenotype. |

#### **NeatEvolutionSettings:**

NeatEvolutionSettings ist ein Hilfs-Struct, welches für die Initialisierung eines NeatEvolutionManager benötigt wird. Es enthält sämtliche einstellbare, relevante Evolutionsparameter.

|  |  |  |
| --- | --- | --- |
| **Parameter** | **Default** | **Beschreibung** |
| uint PopulationSize | 100 | Anzahl an Individuen pro Generation. Eine höhere Anzahl bedeutet eine schnellere Abdeckung des Search Space, wirkt sich jedoch auf die Performance aus. |
| uint SpeciesCount | 10 | Anzahl an Spezies pro Generation. Dies ist ein Richtwert, die tatsächliche Anzahl fluktuiert. |
| uint InterspeciesMatingChance | 0.001 | Die Wahrscheinlichkeit von speziesübergreifender Paarung. HINWEIS: DIESER WERT HAT AKTUELL KEINE WIRKUNG! |

#### **SimplisticGenotype:**

SimplisticGenotype ist C# als Klasse realisiert, folglich sind sämtliche Funktionen Klassenmethoden. Objekte dürfen nicht direkt erstellt werden, Instanzen sind lediglich über die Funktionen von EvolutionManager zu erhalten.

Ein Objekt entspricht einem Handler, welcher auf einen vom EvolutionManager gespeichertes Genotyp verweist. Dementsprechend können diese von gewissen Methoden des EvolutionManagers invalidiert werden, genaueres ist der Sektion *Seiteneffekte*der jeweiligen Funktion zu entnehmen.

**Zugriff auf die aktuelle Fitness:**

|  |  |
| --- | --- |
| **Signatur** | float GetFitness() |
| **Funktion** | Liefert die aktuelle Fitness des Genotyps zurück. |
| **Verwendung** | Kann zum Vergleichen von Genotypen genutzt werden, insbesondere zur Abfrage der Fitness des besten Netzwerks der vorigen Generation. |
| **Argumente** | - |
| **Returns** | Die Fitness als float. Wurde (noch) keine Fitness gesetzt entspricht der Wert 0. |
| **Threadsafe** | Nein |
| **Seiteneffekte** | - |

**Setzen der Fitness:**

|  |  |
| --- | --- |
| **Signatur** | void SetFitness(float t\_fitness) |
| **Funktion** | Ersetzt die aktuelle Fitness mit dem gegebenen Wert. Die Evolution geschieht auf Basis dieser Größe, wobei eine höhere Zahl einem kompetenteren Netzwerk entspricht. |
| **Verwendung** | Schlüsselmethode zur Bewertung von Individuen. Mittels dieser Funktion muss jedes Netzwerk jede einzelne Generation bewertet werden. |
| **Argumente** | *t\_fitness:* Eine nicht-negative Kommazahl, höher bedeutet besser. |
| **Returns** | - |
| **Threadsafe** | Nein |
| **Seiteneffekte** | - |

**Dekodieren:**

|  |  |
| --- | --- |
| **Signatur** | SimplisticPhenotype Decode() |
| **Funktion** | Dekodiert den Genotyp in einen Phänotyp. |
| **Verwendung** | Liefert ein verwendbares neuronales Netzwerk, welches eine Repräsentation des entsprechenden Genotyps darstellt. Dieses kann zur Ermittlung der Fitness im Rahmen einer Aufgabe getestet werden. |
| **Argumente** | - |
| **Returns** | Eine Instanz der Klasse SimplisticPhenotype. |
| **Threadsafe** | Nein |
| **Seiteneffekte** | - |

#### **SimplisticPhenotype:**

SimplisticGenotype ist C# als Klasse realisiert, folglich sind sämtliche Funktionen Klassenmethoden. Objekte dürfen nicht direkt erstellt werden, Instanzen sind lediglich über die Decode-Funktion eines Genotyps zu erhalten.

SimplisticGenotype implementiert ebenfalls das IDisposable-Interface. Dies ist relevant, da die Netze durchaus einen erheblichen Speicherbedarf aufweisen können. Um nicht auf unregelmäßige Garbage-Collection angewiesen zu sein, sollte die Funktion Dispose() aufgerufen werden, sobald das Objekt nicht mehr benötigt wird.

**Dispose:**

|  |  |
| --- | --- |
| **Signatur** | void Dispose() |
| **Funktion** | Gibt die von der Instanz beanspruchten Ressourcen frei. |
| **Verwendung** | Diese Funktion sollte aufgerufen werden, sobald das Objekt nicht mehr benötigt wird, um die RAM-Auslastung zu begrenzen. |
| **Argumente** | - |
| **Returns** | - |
| **Threadsafe** | Nein |
| **Seiteneffekte** | Invalidiert das Objekt. |

**Setzen der Eingänge:**

|  |  |
| --- | --- |
| **Signatur** | void SetInput(ulong t\_index, float t\_value) |
| **Funktion** | Setzt den spezifizierten Eingang auf den gegebenen Wert. |
| **Verwendung** | Vorbereitung des Netzwerks für die Aktivierung. |
| **Argumente** | *t\_index:* Zahl im Bereich *0 <= x < Eingangsanzahl*, spezifiziert den zu setzenden Eingang. Dieser Wert wird auf Validität überprüft.  *t\_value:* Eingangsgröße als Fließkommazahl. |
| **Returns** | - |
| **Threadsafe** | Nein |
| **Seiteneffekte** | - |

**Zugriff auf die aktuellen Eingangswerte:**

|  |  |
| --- | --- |
| **Signatur** | float GetInput(ulong t\_index) |
| **Funktion** | Liefert den aktuellen Eingangswert zurück. |
| **Verwendung** | - |
| **Argumente** | *ulong* *t\_index:* Zahl im Bereich *0 <= x < Eingangsanzahl*, spezifiziert den zu setzenden Eingang. Dieser Wert wird auf Validität überprüft. |
| **Returns** | *float:* Aktuelle Eingangsgröße. |
| **Threadsafe** | Nein |
| **Seiteneffekte** | - |

**Netzwerkaktivierung:**

|  |  |
| --- | --- |
| **Signatur** | void Activate(uint t\_iterations) |
| **Funktion** | Aktiviert das neuronale Netzwerk, dies resultiert in der internen Verarbeitung der Eingänge und entsprechender Aktualisierung der Ausgänge.  Der NEAT-Algorithmus ermöglicht die Bildung interner Rückkopplungen und somit Neuronen mit Speicherfunktion. Die gegebenen Iterationen bestimmen die Anzahl an Berechungsdurchläufen und somit die Signifikanz dieser Schleifen. |
| **Verwendung** | Kernfunktionalität des neuronalen Netzwerks. |
| **Argumente** | *uint* *t\_iterations:* Bestimmt die Anzahl an internen Berechnungsiterationen. Eine höhere Zahl verbessert die Speichermöglichkeiten eines Netzwerkes, wirkt sich jedoch auf die Performance aus. Erfahrungsgemäß sollte dieser Wert 3 nicht unterschreiten, dies ist jedoch Szenario abhängig und sollte individuell experimentell überprüft werden. |
| **Returns** | *-* |
| **Threadsafe** | Nein |
| **Seiteneffekte** | - |

**Reset der Zustände:**

|  |  |
| --- | --- |
| **Signatur** | void Reset() |
| **Funktion** | Setzt den internen Netzwerkzustand und die Ausgänge zurück. |
| **Verwendung** | Die Verwendung dieser Funktion wird vom gewünschten Speichervermögen des Netzwerks bestimmt. Ist die Aufgabe rein kombinatorisch, also ausschließlich von den Eingängen und nicht von inneren Zuständen abhängig, so sollten Reset() vor jeder Aktivierung aufgerufen werden. |
| **Argumente** | *-* |
| **Returns** | *-* |
| **Threadsafe** | Nein |
| **Seiteneffekte** | - |

#### **Beispielhafte Verwendung**

// Settings for Evolution

NeatEvolutionSettings settings = new NeatEvolutionSettings()

{

PopulationSize = 100,

SpeciesCount = 10,

InterspeciesMatingChance = 0.001f

};

// create evolution manager

NeatEvolutionManager manager = new NeatEvolutionManager(2, 1, settings);

// try to load population, if no file exists: create starting population

try

{

manager.Load("population.xml");

}

catch

{

manager.CreateRandomPopulation();

}

uint generation = 0;

while (true)

{

// print generations best fitness via acquiring of best genotype

// the best genotype is updated after calling EvolvePopulation()

float bestFitness = manager.GetPreviousBestGenotype().GetFitness();

Console.WriteLine($"Best fitness: {bestFitness}");

// increase generation and print current number

Console.WriteLine($"\n============== Generation {++generation} ==============");

// evaluate fitness of networks

foreach (SimplisticGenotype genotype in manager.GetPopulation())

{

// decode genotype into phenotype

SimplisticPhenotype phenotype = genotype.Decode();

// set inputs – GetNextInput() is custom function

phenotype.SetInput(0, GetNextInput());

phenotype.SetInput(1, GetNextInput());

// Reset because should be combinatorial network

phenotype.Reset();

// in this case, the (arbitrary) problem doesn’t require internal states, so

// iterations should be kept low

phenotype.Activate(3);

float output = phenotype.GetOutput(0);

// CalculateFitness is a custom function

float fitness = CalculateFitness(output);

genotype.SetFitness(fitness);

// phenotype is not needed anymore – should be done with try – catch in an

// actual program

phenotype.Dispose();

}

// advance evolution by one generation

manager.EvolvePopulation();

// save population every 100 generations

if (generation % 100 == 0)

{

manager.Save("population.xml");

}

}

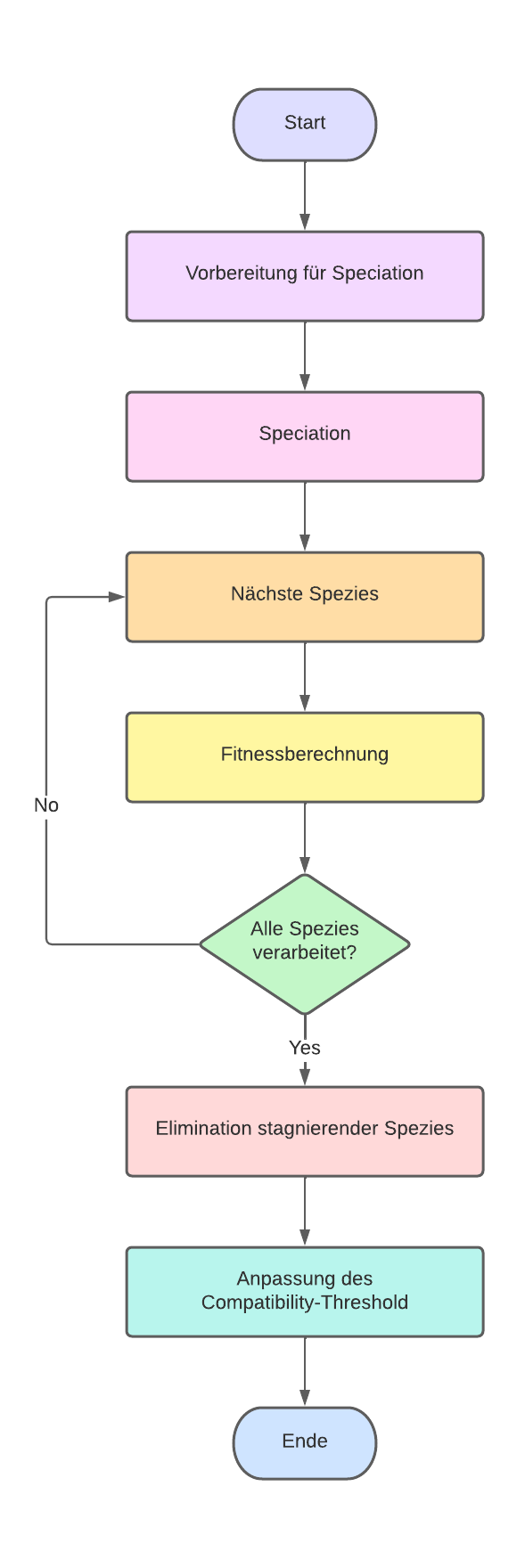
### Details

Die folgende Sektion geht auf einige Schlüsselpunkte bei der Realisierung der komplexeren Hauptkomponenten ein.

#### **EvolutionManager**

Die zwei Hauptaufgaben des EvolutionManagers sind Verwaltung der Individuen und Koordination des Evolutionsablaufes.

Schwierigkeit der ersten dieser beiden Aufgaben ist, dass Genotypen sowohl in einer allgemeinen Populationsliste als auch in den jeweiligen Spezies referenziert werden. Diese Container werden unabhängig voneinander bearbeitet, daher darf die Löschung der Instanzen erst erfolgen, wenn diese nicht mehr benötigt werden. Aus diesem Grund geschieht die Speicherung mittels *Shared Pointers.*

**Spezies-Handling:**

Der erste Schritt der Evolution gilt den Spezies der Population, insbesondere der Aufteilung der Genotypen auf diese.

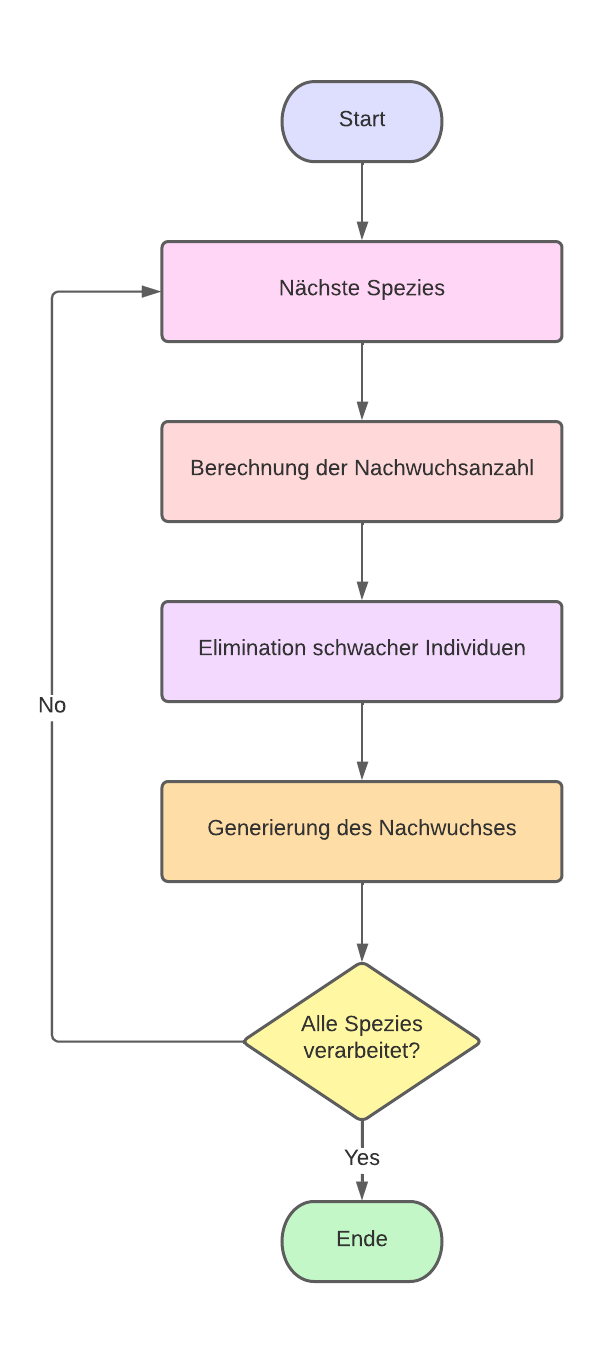
Die Vorbereitung besteht hauptsächlich aus dem Zurücksetzen der Spezies der vorhergehenden Generation. Die Besonderheit ist allerdings, dass erfolgreiche Spezies in die nächste Generation übernommen werden. Erfolgreich ist eine Spezies, wenn sie überdurchschnittlich viele Individuen enthält, also wenn gilt , wobei n die Anzahl an Individuen und NSpezies die Zielanzahl an Spezies darstellt.

Um Spezies auch ohne Individuen übernehmen zu können, verfügt die jede Spezies über ein temporäres Referenz-Objekt, das zur Distance-Berechnung genutzt wird, dieses entspricht dem vorigen Champion.

Anschließend werden die Genotypen aufgeteilt, diese werden nach Möglichkeit einer existierenden Spezies zugewiesen. Ist nirgendwo Kompatibilität gegeben, wird eine neue Spezies erstellt.

Als Nächstes folgt die Berechnung der Speziesfitness gemäß der im Kapitel 2.3.4 angeführten Formeln. Es folgt eine Überprüfung, ob die Spezies seit mehr als 15 Generationen stagniert, also sich ihre Fitness nicht verbessert hat. Ist dies der Fall, so werden die Spezies und die ihr zugeordneten Individuen entfernt.

Schlussendlich wird der Compatibility-Threshold entsprechend dem Unterschied zwischen Spezies Soll- und Ist-Anzahl angepasst.

**Nachwuchserzeugung:**

Im nächsten Schritt geschieht die eigentliche Produktion der Nachkommen, hierzu werden die Spezies in einer Schleife abgearbeitet.

Zuallererst wird die Anzahl festgelegt, dies geschieht entsprechend der in Kapitel 2.3.4 beschriebenen Formeln. Es ist von entscheidender Bedeutung, diese Berechnung erst *nach* der Elimination stagnierender Spezies durchzuführen, da hierbei die Fitnesswerte der einzelnen Gruppen aufsummiert werden und nicht fortpflanzungsgenehmigte Spezies diesen Wert verfälschen würden.

Als nächstes erfolgt die Elimination der Individuen mit niedriger Fitness, konkret wird nur den fähigsten 25% der Genotypen die Fortpflanzung erlaubt, diese kompetenten Netzwerke dienen nun als Basis zur Fortpflanzung.

**Fortpflanzung:**

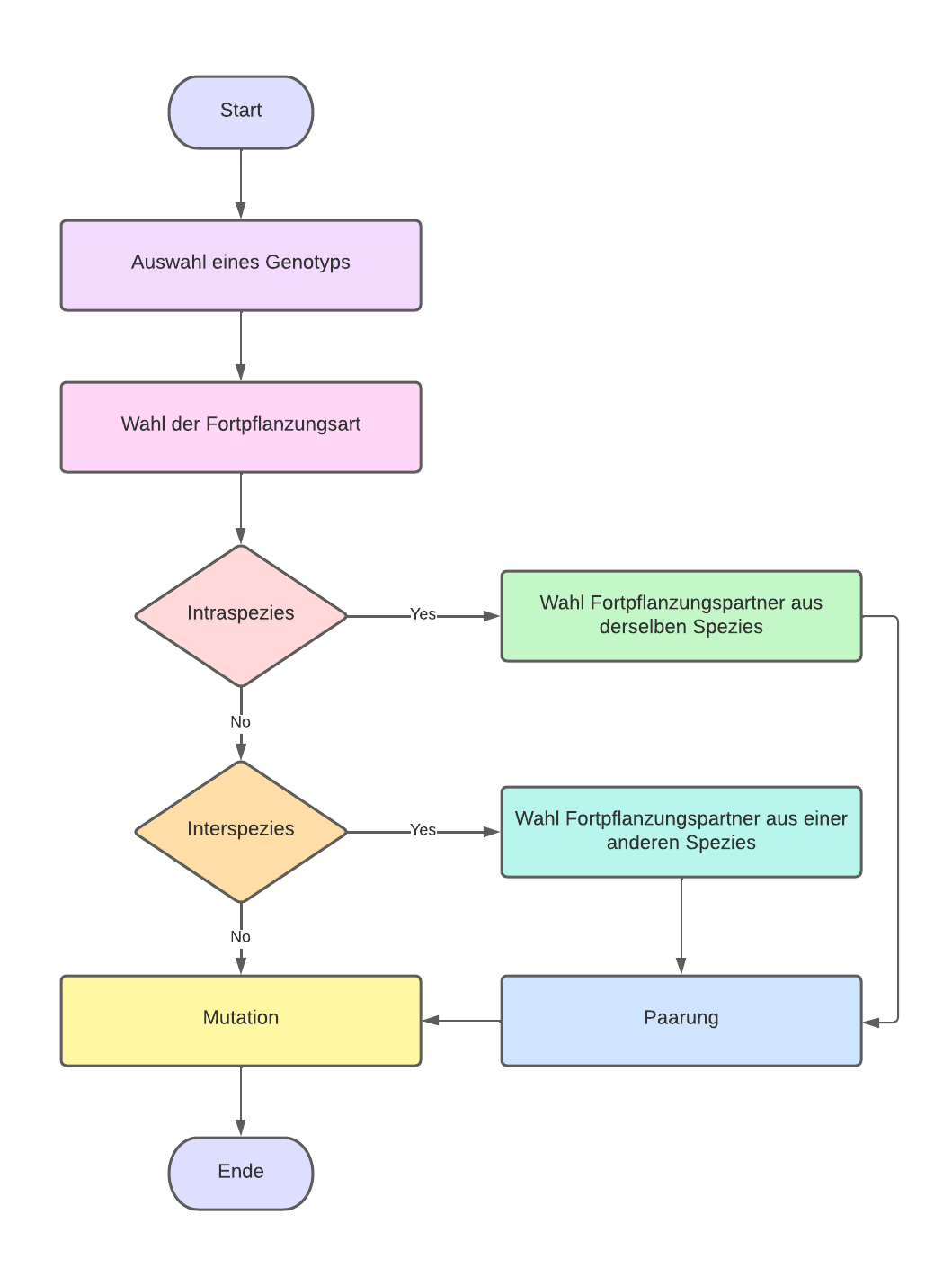


Abbildung visualisiert den Mechanismus der Fortpflanzung. Wurde ein Individuum gewählt, erfolgt die zufällige Auswahl einer der drei folgenden Varianten:

* **Intraspecies Mating** bezeichnet die Paarung mit einem kompatiblen Partner, also einen Genotyp aus derselben Spezies. Diese Option ist mit 75% am wahrscheinlichsten.
* **Interspecies Mating** ist die speziesübergreifende Paarung. Diese sorgt für Diversität, allerdings ist die Chance auf ein funktionstüchtiges Nachkommen deutlich hierbei geringer als bei der vorigen Variante. Aus diesem Grund beträgt die Wahrscheinlichkeit hierfür lediglich 0.1%.
* **Keine Paarung**

Unabhängig von den vorigen Optionen wird jeder Genotyp mutiert.

Die Wahrscheinlichkeitswerte richten sich nach den Werten im originalen NEAT-Paper.

#### https://documents.lucid.app/documents/e28ccfa8-9da1-4403-9314-19322f98bc84/pages/0_0?a=2274&x=537&y=-1938&w=617&h=1430&store=1&accept=image%2F*&auth=LCA%20c2e8a2e04726fcd99754bbd9e948cb882b6f5b42-ts%3D1648209099**SimplisticGenotype**

Die Hauptaufgabe der Genotyp-Klasse liegt in der Fortpflanzung, konkret in den Teilbereichen Mating und Mutation.

**Mating:**

Mating oder Crossoverbedeutet die Fortpflanzung zweier Genotypen. Hierbei agiert das fittere Elternteil als Vorlage für das Nachkommen, vererbt folglich sämtliche eigenen Genome.

Anschließend wird durch die Genome des zweiten Genotyps iteriert und diese mit der Vorlage verglichen. Finden sich zwei Genome mit derselben Innovation-Number in beiden Eltern, so wird das Weight der Vorlage durch die Durchschnittsgewichtung ersetzt.

Haben beide Eltern dieselbe Fitness, so werden Genome mit einer Wahrscheinlichkeit von 50% übernommen.

**Mutation:**

Die Genotypen sind außerdem für Mutationen verantwortlich, dies geschieht durch die folgenden Operationen:

* **Weight-Mutation** einer einzelnen Verbindung mit einer Wahrscheinlichkeit von 80%.
* **Hinzufügen einer Verbindung** mit zufälligem Source, Target und Weight mit einer Wahrscheinlichkeit von 5%.
* **Hinzufügen eines Neurons** durch Spaltung einer existierenden Verbindung. Die erste Verbindung übernimmt hierbei die Gewichtung der ursprünglichen, während die zweite ein Weight von 1 erhält. Dies führt zu einer Nichtlinearität aufgrund der Aktivierungsfunktion. Dies geschieht mit einer Wahrscheinlichkeit von 3%.
* **Entfernen einer Verbindung**, momentan ungenutzt.

Die Wahrscheinlichkeit der Mutationsoperatoren sind unabhängig voneinander, die Wahrscheinlichkeiten wurden großteils der originalen NEAT-Implementation entnommen. Die Enable-Bit-Funktionalität wurde nicht inkludiert.

Zusätzlich unterstützt die Implementation mutierende Evolutionsparameter. Bei dieser Variante werden die relevanten Wahrscheinlichkeiten generationsweise zufällig leicht verändert, um der Evolution die Anpassung an das Trainingsszenario zu ermöglichen.

**Genomverwaltung:**

Um den Prozess der Fortpflanzung zu vereinfachen, werden die Genome intern nach Innovation-Number sortiert gespeichert. Dies ermöglicht einen schnelleren Vergleich mit einer Komplexität von O(n). Allerdings erfordern die Mutationen auch die Auswahl eines zufälligen Genoms sowie eine schnelle Löschung.

Aktuell werden die Genome allerdings als Vektor gespeichert, wodurch die Delete-Operation eine Komplexität von O(n2) aufweist.

#### https://documents.lucid.app/documents/e28ccfa8-9da1-4403-9314-19322f98bc84/pages/0_0?a=2703&x=2307&y=-2435&w=684&h=2080&store=1&accept=image%2F*&auth=LCA%20fa4b0399b8ff5c10be12f43763dbce15f3f2d286-ts%3D1648209099 **SimplisticPhenotype**

Die Funktion des SimplisticPhenotype besteht in der Berechnung des Ausganges anhand der Eingänge.

Hierfür verfügt es über nach Zielneuronen geordneten Verbindungen, sowie einen Vektor der Neuronenzustände.

Wird das Netzwerk nun aktiviert, so iteriert eine Schleife über sämtliche Hidden- und Output-Neuronen. Der Zustand des aktuellen Neurons wird als Erstes zurückgesetzt, anschließend erfolgt die Iteration über die Verbindungen zu den Eingängen.

In dieser zweiten Schleife werden die Zustände des Source-Neurons ausgelesen und mit dem Weight multipliziert. Die resultierenden Signale sämtlicher Eingänge werden anschließend aufsummiert, der Ausgang ergibt sich mittels Aktivierungsfunktion. (Beispielsbild?)

Die beiden Schleifen werden solange durchlaufen, bis das Iterationslimit erreicht wurde.

Die benötigte Zeit dieses Algorithmus ist direkt proportional zum Iterationslimit, allerdings besteht die Möglichkeit verschiedener Optimierungen.

#### **Decoding**

Dieser Abschnitt beschreibt die spezifische Funktionsweise der Dekodierung von SimplisticGenotype zu SimplisticPhenotype. Diese besteht hauptsächlich aus dem Umsortieren der Verbindungen. Während der Genotyp über nach Innovationsnummer sortierte Genome verfügt, benötigt der Phänotyp nach Zielneuron geordnete Verbindungen. Die Sortierung erfolgt in einen zweidimensionalen Vektor.

### Probleme

**Design:**

Das aktuelle Design weist folgende drei bedeutende Schwächen auf:

* Eine Reihe von Komponenten verletzt das **Single Responsibility Principle (SRP).** Dieses sagt aus, dass jede Klasse lediglich eine einzige Aufgabe haben sollte, um Modularität, Verständlichkeit und Wartbarkeit zu maximieren. Dieses Problem ist durch eine Aufspaltung der Klassen und Verantwortungen jedoch relativ einfach zu lösen.
* Ein weiteres Problem stellt die Verwendung von statischen Variablen zur Speicherung von der maximalen Innovation- und Neuron-Number. Dies führt zu undefiniertem Verhalten, wenn mehrere Instanzen des EvolutionManagers erstellt werden.  
  Außerdem werden unbenutzte IDs nicht zur erneuten Nutzung freigegeben, was bei entsprechend langer Laufzeit zwangsläufig zu einem Overflow führt.
* Außerdem verfügt die aktuelle Implementation der SimplisticGenotype-Klasse nicht über Neuronen-Gene, sondern lediglich über Verbindungen. Aufgrund dieser Tatsache ist es nicht möglich, unterschiedliche Aktivierungsfunktionen oder Eingangsverarbeitungen zu verwenden.

#### **Performance:**

Die Implementation weißt einige erhebliche Performance-Probleme auf. Diese sind in erste Linie auf für ihre Aufgabe unzureichende Container zurückzuführen:

* Die Genom-Verwaltung des SimplisticGenotype verwendet aktuell einen Vektor, welcher eine quadratische Worst-Case Delete-Komplexität aufweist (siehe Kapitel 2.4.4.1). Dieses Problem könnte durch die Implementation eines speziellen GenomeContainers mit den entsprechenden Eigenschaften behoben werden.
* Die SimplisticPhenotype-Klasse sortiert Verbindungen und Neuronenzustände nach Neuronennummer, wobei diese dem Index entspricht. Allerdings verwenden die einzelnen Genotypen nur einen winzigen Bruchteil aller Neuronen, wodurch diese Methode in Extremfällen einen Overhead um den Faktor 500+ aufweist. Dieses Problem könnte durch einen Mapping-Approach gelöst werden, bei dem sich Neuronennummern von Genotyp und Phänotyp unterscheiden.
* Der aktuelle Decoding-Ansatz wird ebenfalls durch die quadratische Delete-Komplexität von Vektoren limitiert.

Außerdem existiert momentan keine Unterstützung für reine Feedforward-Netzwerke, also Netze ohne interne Rückkopplungen und somit ohne Speicherfähigkeit. Eine Implementation einer solchen Option könnte die Aktivierung eines Phänotyps deutlich beschleunigen, nicht zuletzt, da lediglich eine einzige Iteration erforderlich wäre.

## Verkehrssteuerung mittels neuronalen Netzwerks

### Konzept

Für die Verkehrssteuerung verfügt das neuronale Netzwerk über die vollständige Kontrolle sämtlicher Fahrzeuge. Erreicht ein Verkehrsteilnehmer eine Kreuzung, so sendet er eine Richtungsanfrage, welche die KI mit einer der vier Himmelsrichtungen beantwortet.

Das Straßennetz wird als Raster in Form zweier Listen repräsentiert: Eine Existenz-Bitmask indiziert, ob eine Straße vorhanden ist, eine zweite Liste enthält die Geschwindigkeitsinformationen, welche sich aus Straßenkapazität und Auslastung zusammensetzen.

Eine Trennung dieser beiden Informationen erfolgt, da eine niedrigere Geschwindigkeit und eine inaktive Straße fundamental unterschiedlich sind. Ein neuronales Netzwerk funktioniert jedoch mittels Interpolation, kann also nur schwer zwischen einer sehr niedrigen Geschwindigkeit und einer nicht-vorhandenen Verbindung unterscheiden. Dies kann bei einem nur über eine schlechte Straße erreichbaren Zielpunkt zum Problem werden.

Eine weitere Liste mit derselben Länge kommuniziert Start- und Ziel des aktuellen Fahrzeugs. Die aktuelle Position wird mit 1 markiert, das Ziel mit -1. Die übrigen Indizes enthalten den Wert 0.

### Ein- und Ausgänge

|  |  |  |
| --- | --- | --- |
| **Typ** | **Beschreibung** | **Anzahl** |
| Eingang | **Existenzmaske:** Zeigt an, ob eine Straße existiert. | Straßenanzahl |
| Eingang | **Geschwindigkeitsmaske:** Entspricht der maximalen Geschwindigkeit, welche auf der Straße momentan erreicht werden kann. | Straßenanzahl |
| Eingang | **Routenmaske:** Gibt Ziel mit -1 und aktuelle Position mit 1 an. | Straßenanzahl |
| Ausgang | Bestimmt die weitere Fahrtrichtung. Jeweils ein Ausgang steht für eine der vier Himmelsrichtungen, der höchste Wert entscheidet die Route. | 4 |

### Fitnessberechnung

Die Fitness setzt sich aus drei Komponenten zusammen:

* **Zeitdauer-Fitness:** Wie schnell wird das Ziel erreicht?
* **Validitäts-Fitness:** Wie oft ist das Fahrzeug valide abgebogen?
* **Entfernungs-Fitness:** Wird das Ziel nicht erreicht, wie weit ist der Endpunkt vom Ziel entfernt?

Der Grund für die Betrachtung dieser drei Teilaspekte der Fitness liegt vor allem am Trainingsprozess. Eine Erreichung des Ziels ist in den frühen Stadien nicht zu erwarten, dennoch muss eine Verbesserung der Routenfindung in der Fitness festgehalten werden. Hierfür sind Validitäts- und Entfernungs-Fitness verantwortlich.

Die **Zeitdauer-Fitness** belohnt schnelles Erreichen des Ziels. Wird das Zeitlimit überschritten oder gibt das Netzwerk eine invalide Routenangabe, so beträgt dieser Wert 0. Berechnet wird diese Größe nach folgender Funktion:

Die resultierende Funktion (Abbildung) entspricht einer homogen fallenden Polynomialfunktion 3. Grades. Für den konkreten Anwendungsfall gilt , wodurch ein Nulldurchgang bei tnorm = 1 existiert und die maximalen Fitness von 10 bei t­norm = 0 erreicht ist. Der Wendepunkt liegt bei tnorm = 0.5.

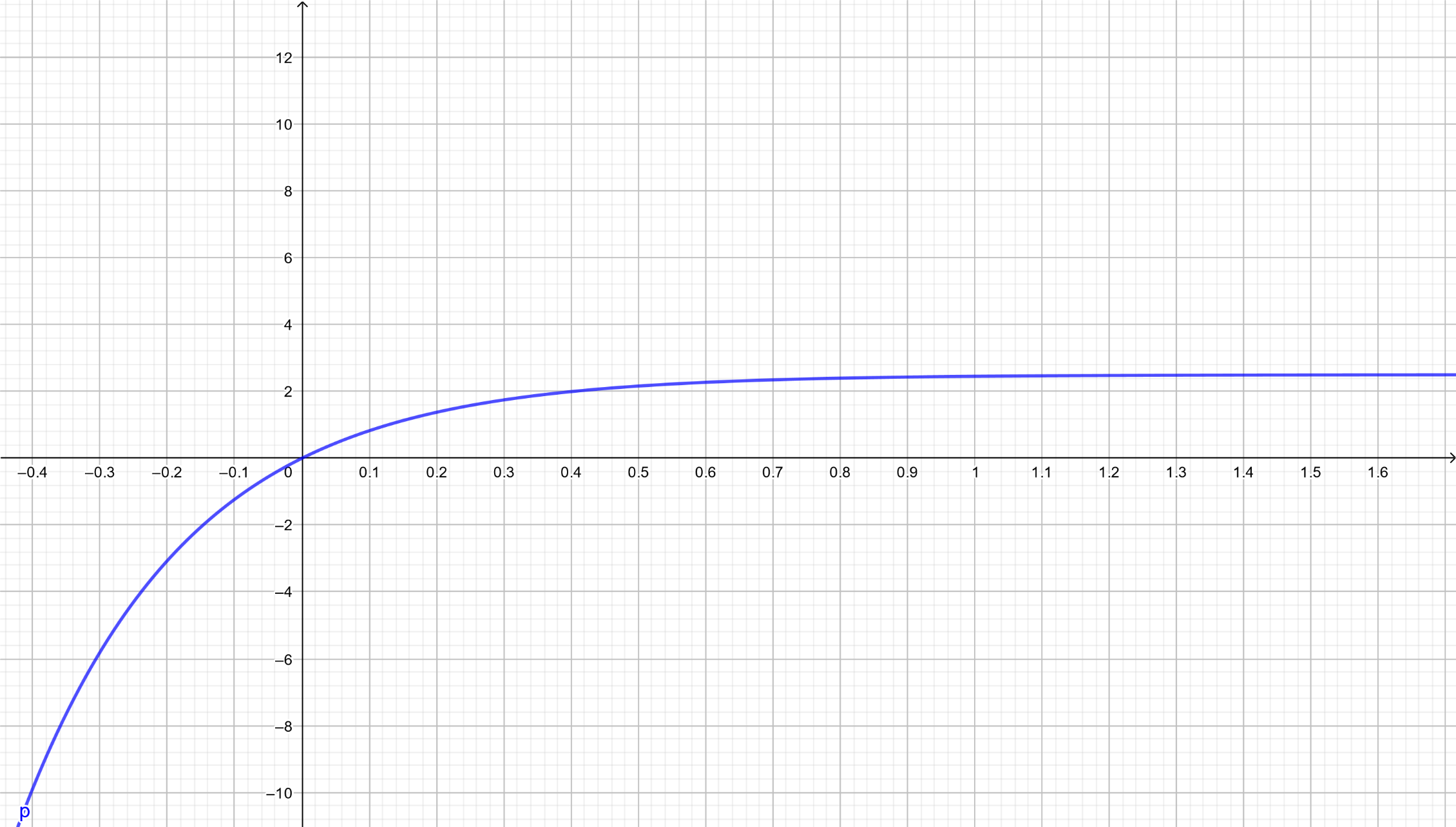


Diese Funktion wurde aufgrund der zum rechten Ende hin stärkeren Steigung gewählt. Dies soll die Streuung der Fahrzeiten verringern, also die Bevorzugung einiger weniger Fahrzeuge unattraktiver machen. Die Idee dahinter ist, dass eine Zeitreduzierung sehr langsamer Verkehrsteilnehmer verhältnismäßig einfach ist, dabei aber einen hohen Fitnesszuwachs verursacht.

Als zweiter Faktor fällt die **Validität** der Routenangaben ins Gewicht. Als valide gilt eine Routenangabe, wenn in der angewiesenen Richtung eine befahrbare Straße existiert. Dieser Teil der Fitness ist hauptsächlich für die frühen und mittleren Trainingsphasen gedacht, wenn das Ziel noch nicht erreicht wird.

Auch diese Berechnung wird normiert, in diesem Fall auf die maximale Anzahl an Routenänderungen, welche durch das Zeitlimit gegeben ist:

Der Buchstabe *c* steht hierbei für *choice.*

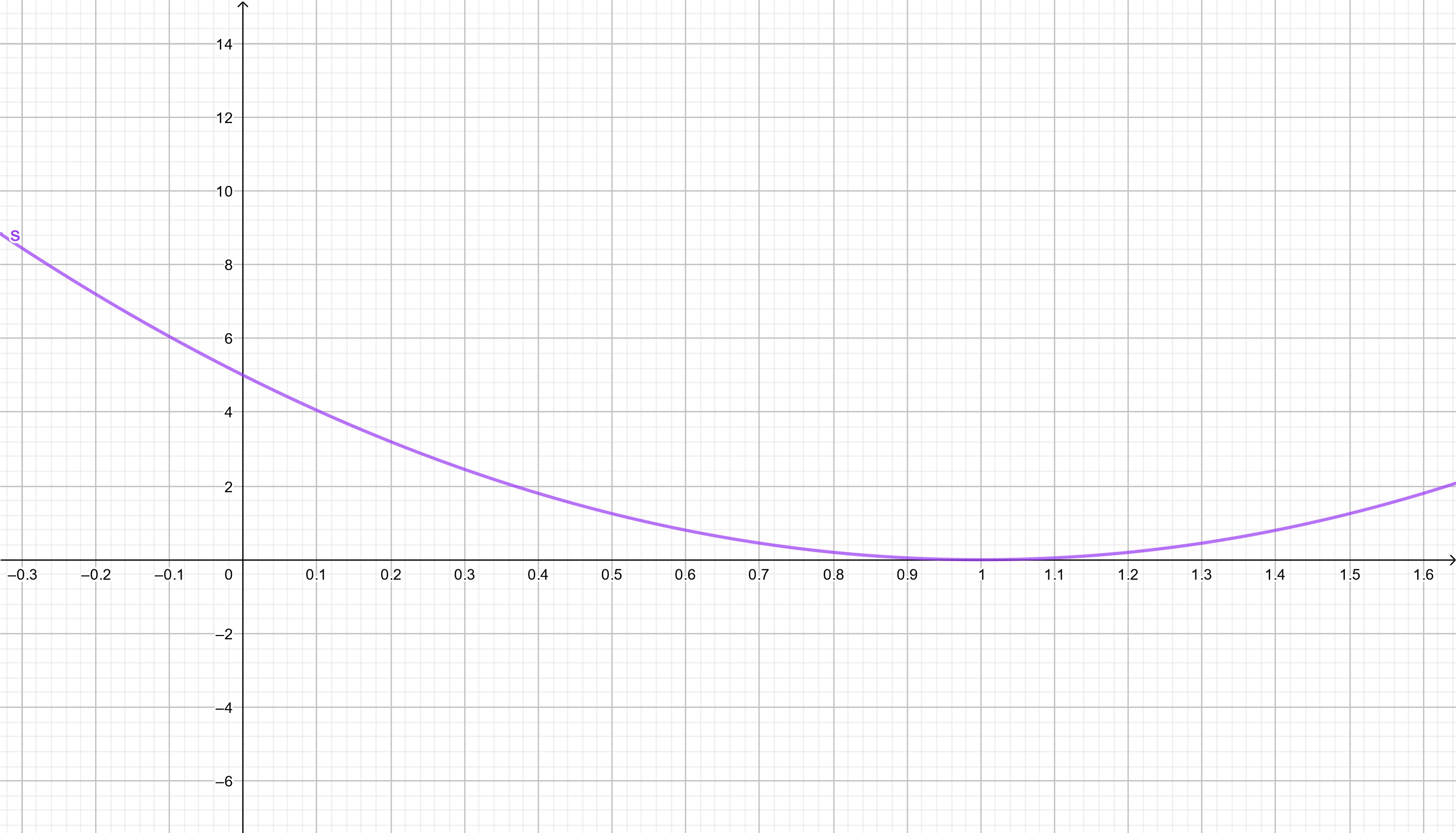


Die Definitionsmenge ist , die Nullstelle liegt bei fc(0), der Maximalwert beträgt fc (1) ~ 2.5.

Auch ist die Steigung dieser Funktion immer kleiner als die der Zeit-Fitness, weshalb eine Verkürzung der Zeit priorisiert wird. Zur Vermeidung von Fahrzeugen, welche ausschließlich im Kreis fahren, ist dies von entscheidender Bedeutung.

Außerdem spielt die **Entfernung** zwischen erreichtem Punkt und Ziel eine Rolle. Diese soll das Netzwerk während des Trainingsprozesses motivieren, die Fahrzeug in Richtung des Zieles zu dirigieren, auch wenn es dieses noch nicht erreicht. Der Abstand wird auf die Diagonale des Rasters normiert, folglich gilt und .

Die Fitness wird nach einer quadratischen Funktion ermittelt (Abb):



Für die Definitionsmenge gilt erneut , der Maximalwert beträgt 5 bei . Damit ist die Entfernung deutlich wichtiger als valide Routenangaben, die schwache Steigung im linken Bereich soll es jedoch möglich machen, sich ohne zu großen Fitnessverlust vom Ziel zu entfernen, falls kein direkter Weg existiert. Außerdem wirkt dieser Teil der Fitness wie ein Zielerreichungs-Bonus, selbst wenn dies in letzter Sekunde geschieht, die Zeitfitness also 0 beträgt.

Die Maximalfitness ist vom jeweiligen Straßennetz abhängig, da der beste Weg in unterschiedlichen Szenarien unterschiedlich lang ist und folglich eine andere Zeit in Anspruch nimmt.

Aus diesem Grund und da Generierung des Straßennetzes und Routenwahl rein zufallsbasiert erfolgen, ist eine größere Anzahl an Testfahrzeugen ausschlaggebend, um den Zufallsfaktor zu minimieren.

### Erweiterungsmöglichkeiten

In Zukunft ist eine Erweiterung denkbar, welche dem Netzwerk ermöglicht, dichten Verkehr vorauszusagen und zu vermeiden. Mit einer solchen Ergänzung übernimmt der Algorithmus nicht mehr nur die Aufgabe der Wegfindung, sondern auch den Auftrag, die einzelnen Verkehrsteilnehmer möglichst effizient an ihr jeweiliges Ziel zu bringen. Hierzu wird die Auswirkung von dichtem Verkehr in der Straßennetzsimulation berücksichtigt: Je mehr Autos sich auf derselben Straße befinden, desto langsamer wird ihre Geschwindigkeit. Dies hat zum Ziel, das neuronale Netzwerk zu einer Aufteilung der Fahrzeuge auf verschiedene Routen zu animieren.

Für die Umsetzung wird ein Eingang für jede mögliche Start-Ziel-Kombination hinzugefügt, welcher einen Wert entsprechend der Anzahl der Fahrzeuge auf dieser Route enthält. Dies befähigt die KI, Annahmen über die Verkehrsdichte auf den einzelnen Straßen zu treffen und die Routen dementsprechend zu ändern.

Um zwischen einzelnen Verkehrsteilnehmer auf derselben Strecke unterscheiden zu können, muss das Netzwerk zusätzlich mit einem *Numbering-Input* ausgestattet werden, welcher für jedes Fahrzeug auf derselben Route eine eigene ID ausgibt.

### Komplexität

Komplexität bezieht sich in diesem Fall auf die Anzahl der Ein- und Ausgänge (NE und NA) im Verhältnis zur Straßenanzahl *n*.

Die einfache Wegfindung weist eine konstante Ausgangsanzahl mit NA = 4 sowie eine lineare Eingangskomplexität nach folgender Formel auf:

Jede der drei Eingangsmasken (siehe Kapitel 2.5.2) hat dieselbe Größe n.

Im Erweiterungsszenario ist die Nummer der Ausgänge ebenfalls mit NA = 4 konstant, die Zahl der Eingänge steigt jedoch quadratisch mit der Straßenanzahl:

Dieser Term setzt sich aus den drei Eingangsmaske (3n), dem *Numbering-Input* (1) sowie der möglichen Routen (n2 + n) zusammen.

Die Anzahl der Start-Ziel-Kombinationen NR berechnen sich folgendermaßen:

Die Summenformel berechnet die Zahl der Kombinationen ungeachtet ihrer Reihenfolge, da eine Route jedoch in beiden Richtungen befahren werden kann, erfolgt eine Multiplikation mit dem Faktor 2.

Mittels der Gaußschen Summenformel kann die Summe auch folgendermaßen angeschrieben werden:

Aufgrund der quadratischen Komplexität wäre bei der zweiten Variante eine starke Limitierung der Straßennetzgröße nötig, im weiteren Verlauf wird jedoch nur das einfache, nicht-erweiterte Layout verwendet.

# Erklärung der Eigenständigkeit der Arbeit

EIDESSTATTLICHE ERKLÄRUNG

Ich erkläre an Eides statt, dass ich die vorliegende Arbeit selbständig und ohne fremde Hilfe verfasst, andere als die angegebenen Quellen und Hilfsmittel nicht benutzt und die den benutzten Quellen wörtlich und inhaltlich entnommenen Stellen als solche erkenntlich gemacht habe. Meine Arbeit darf öffentlich zugänglich gemacht werden, wenn kein Sperrvermerk vorliegt.

|  |  |  |
| --- | --- | --- |
| Ort, Datum |  | Verfasser 1 |

|  |  |  |
| --- | --- | --- |
| Ort, Datum |  | Verfasserin 2 |

…

# Abbildungsverzeichnis

[Abbildung 1: XYZ vi](#_Toc415201720)

[Abbildung 2:ABC [VI](#_Toc415201721)**.**](#_Toc415201721)

# Tabellenverzeichnis

[Tabelle 1: Arbeitsaufstellung IV](#_Toc415201490)

# Literaturverzeichnis

# Abkürzungs- und Symbolverzeichnis

Anhang

# A1 Pflichtenheft (OPTIONAL)

A2 Schlussfolgerung / Projekterfahrung

A3 Projektterminplanung

## A4 Arbeitsnachweis Diplomarbeit

A5 Datenblätter (OPTIONAL)

A6 Technische Zeichnungen (OPTIONAL)

1. vgl. Tom Mitchell 1997, Preface [↑](#footnote-ref-1)
2. vgl. Leslie Pack Kaelbling, Michael L. Littman und Andrew W. Moore 1996, Abstract [↑](#footnote-ref-2)