UNIKASSEL
VERSITÄT

Evaluierung genetischer Programmierung im Kontext eines Tower-Defense Spiels in Haskell

Bachelorarbeit

im Bachelorstudiengang Informatik am Fachbereich 16 Elektrotechnik/Informatik der Universität Kassel, vorgelegt am 30. März 2011 von

Frederik Kreckler
Matrikelnummer: 26206655

Betreuer: Dipl.-Inf. Michael Lesniak

Erstgutachter: Prof. Dr. Claudia Fohry
Zweitgutachter: Prof. Dr. Gerd Stumme

Selbständigkeitserklärung

Ich versichere hiermit, dass ich die Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe.

Kassel, den 30. März 2011

Frederik Kreckler

Inhaltsverzeichnis

1. Einleitung

Die Komplexität der natürlichen Evolution [Dar59] spiegelt sich wider in den zahlreichen Anwendungsmöglichkeiten der evolutionären Algorithmen in der Informatik. Dabei handelt es sich um einen Oberbegriff, unter dem Optimierungsverfahren zusammengefasst werden, die sich bei der Lösungssuche an der Biologie orientieren. Ziel der Entwicklung solcher Algorithmen ist es, die schöpferischen Vorgänge der Natur in Computerprogrammen nachzubilden und zur Lösung komplexer Problemstellungen einzusetzen.

Die vorliegende Arbeit beschäftigt sich mit der genetischen Programmierung, einem Teilgebiet der evolutionären Algorithmen.

Bei der genetischen Programmierung werden Individuen erzeugt, die als eigenständige Programme betrachtet werden. Jedes dieser Programme wird daraufhin untersucht, ob es ein gegebenes Problem lösen kann. Die genetische Programmierung eignet sich insbesondere für Probleme, die eine hohe Komplexität aufweisen [ES03]. Dazu gehört auch die Suche nach Lösungen in großen Suchräumen.

Als spezifische Problemstellung sollte hier das Verhalten der Individuen in einer Simulation untersucht werden, die an Computerspiele nach dem Tower-Defense Prinzip angelehnt ist. Dabei handelt es sich um ein Strategiespiel in dem es das Ziel ist, die Gegner vom Überqueren des Spielfelds abzuhalten. Dazu kann der Spieler Hindernisse auf dem Spielfeld positionieren.

In den bekannten Umsetzungen des Genres werden für die Bewegungen der Individuen verschiedene Wegfindungsalgorithmen eingesetzt, um diesen ein intelligent wirkendes Verhalten zu verleihen. Der Nachteil von Wegfindungsalgorithmen in diesem Kontext ist, dass der kürzeste Weg über das Spielfeld immer als optimal angesehen wird.

Strategisch betrachtet wird es aber in vielen Fällen längere Wege geben, die dennoch die Gewinnchance der Gegner erhöhen und dafür sorgen können, dass das Verhalten der Gegner intelligenter wirkt.

Es sollte untersucht werden, wie sich der Einsatz der genetischen Programmierung auf das Verhalten der Spielgegner auswirkt. Zudem wurden die entwickelten genetischen Bestandteile der Simulation so allgemein gehalten, dass sie auch auf andere Probleme angewendet werden können. Dies wurde durch die Entwicklung eines Frameworks ermöglicht.

Die theoretischen Grundlagen und eine kurze Einführung in die genetische Programmierung werden in Kapitel 2 erläutert. Die praktische Anwendung und Umsetzung erfolgt anschließend in Kapitel 3. Für die Darstellung der Ergebnisse wurde eine grafische Oberfläche erstellt, die in Kapitel 4 beschrieben wird. In Kapitel 5 werden die durchgeführten Messungen aufgeführt und die Ergebnisse ausgewertet.

2. Grundlagen

2.1. Die Programmiersprache Haskell

Haskell ist eine funktionale Programmiersprache, die bereits seit mehr als 20 Jahren eingesetzt wird.

Im Gegensatz zu imperativen Programmiersprachen bestehen in Haskell geschriebene Programme aus seiteneffektfreien Funktionen. Seiteneffekte können in der Programmierung zu unerwarteten und schwer auffindbaren Fehlern führen. Wird auf Seiteneffekte verzichtet, kann diese Fehlerquelle vermieden werden.

Haskell verwendet eine verzögerte Auswertungsstrategie für Ausdrücke, die sogenannte *lazy evaluation*, bei der eine Berechnung erst zu dem Zeitpunkt durchgeführt wird, zu dem ihr Ergebnis tatsächlich notwendig ist. Auf diese Weise ist es in Haskell möglich, Funktionen aufzustellen, die nur teilweise ausgewertet werden. Dies ist besonders dann hilfreich, wenn eine vollständige Auswertung aus Zeit- oder Ressourcengründen nicht möglich wäre. So werden auch Listen mit potenziell unendlich vielen Elementen ermöglicht.

Die starke Typisierung (*strong typing*) von Haskell verbietet das implizite Konvertieren von Typen. Da deswegen alle Typkonvertierungen konkret angegeben werden müssen, werden diesbezügliche Programmierfehler schon zum Zeitpunkt des Kompilierens erkannt [Hut07].

Das nachfolgende Listing soll einen Eindruck vom allgemeinen Aufbau und der Syntax eines in Haskell geschriebenen Programms vermitteln. Es zeigt ein Programm, das mit Hilfe des Quicksort-Algorithmus [Hoa62] zwei Listen mit Zahlen sortiert und das Ergebnis auf der Konsole ausgibt.

```
module Main where

main :: IO ()
main = do
  let sorted1 = qsort [5,2,42,1,52,49]
      sorted2 = qsort [7,3,55,90,1041,85,363]
  putStrLn $ "Sorted1 = " ++ show sorted1
  putStrLn $ "Sorted2 = " ++ show sorted2


qsort :: Ord a => [a] -> [a]
qsort []     = []
qsort (x:xs) = qsort le ++ [x] ++ qsort g
                    where
                       le = [y | y <- xs, y <= x]
                       g  = [y | y <- xs, y > x]
```

Listing 2.1: Beispiel für die Syntax von Haskell

2.2. Genetische Programmierung

Die Technik der genetischen Programmierung gehört zur Gruppe der evolutionären Algorithmen [ES03]. Diese versuchen, die in der Natur ablaufende Evolution von Lebewesen in allgemeine Algorithmen zu formalisieren. Dabei werden aus der Biologie bekannte Abläufe auf ein algorithmisches Problem projiziert. Im speziellen Fall der genetischen Programmierung werden die erzeugten Individuen als eigenständige Programme betrachtet, welche ein vorgegebenes Problem nach bestimmten Kriterien möglichst gut lösen sollen.

Die Umsetzung der genetischen Programmierung kann in mehrere, teilweise optionale, Schritte aufgeteilt werden:

1. Initialisierung

   Zu Beginn der Simulation erfolgt zunächst eine Initialisierung. Dazu wird eine zufällige Gruppe von Initial-Programmen gebildet, die als Ausgangspunkt für die genetischen Operationen dient.

2. Evaluierung

   Anschließend wird für jedes Programm ein Wert berechnet, der angibt, wie gut das Programm das Problem lösen kann. Dabei können beliebige Kriterien angewendet werden, um die Leistung eines Programms zu bewerten.

   Dieser sogenannte Fitnesswert macht es möglich, die Programme nach ihrer Fähigkeit das gesetzte Ziel erreichen zu können zu sortieren, was später bei der Selektion notwendig wird.

3. Selektion

   Mit Hilfe des Fitnesswertes können die besten Programme für die Fortpflanzung ausgewählt werden. Die Anweisungen aus denen ein Programm besteht werden dabei auch als Gene bezeichnet. Im einfachsten Fall erfolgt diese Selektion durch Entfernen der schwächsten Programme. Ein Programm gilt als schwach, wenn es verglichen mit den anderen Programmen einen geringen Fitnesswert aufweist. Es können dabei beliebig viele schwache Programme entfernt werden. Zu den komplexeren Selektionsmethoden gehört die *Roulette-wheel selection* (siehe Abschnitt 3.3.2), bei der die Individuen durch Drehen eines virtuellen Glücksrads ausgewählt werden. Die Fitness der Individuen bestimmt dabei die Größe ihres Segments auf dem Glücksrad, erfolgreichere Individuen werden also häufiger ausgewählt.

   Eine andere Form der Selektion ist die *Tournament selection*, bei der zufällig ausgewählte Individuen miteinander verglichen werden. Das Individuum mit der besseren Fitness wird für die nächste Generation ausgewählt.

Das Ziel dieser Methoden ist es, starke Individuen zu bevorzugen, ohne dabei die schwachen Individuen vollständig zu verdrängen, da diese zu einer größeren Diversität der Lösungsansätze führen können.

4. Rekombination

   Nachdem die Individuen für die nächste Generation ausgewählt wurden, werden findet eine Rekombination statt. Dieser Vorgang wird auch als *Crossover* bezeichnet. Dazu wird jedes Individuum mit einem zufälligen anderen Individuum zu einem Paar zusammengefasst. Nun werden die Gene der beiden Individuen an zufälligen Stellen miteinander vertauscht, so dass zwei neue Individuen entstehen, die in die neue Generation übergehen werden. Ziel dieser Maßnahme ist es, die Stärken unterschiedlicher Individuen kombinieren zu können, um neue, effektivere Lösungen hervorzubringen.

5. Mutation

   Anschließend werden die neu gewonnenen Individuen zufällig mutiert. Mit einer bestimmten Wahrscheinlichkeit werden dabei einzelne Gene eines Individuums zufällig verändert. Auf diese Weise werden neue genetische Strukturen erzeugt, die zur Lösungsfindung beitragen können.

2.3. Das Prinzip des Tower Defense

Tower Defense ist die Bezeichnung für eine Gruppe von Strategiespiel (*Real Time Strategy Game*), bei dem es darum geht, mit Hilfe von verschiedenen Hindernissen eine Gruppe von gegnerischen Einheiten aufzuhalten. Bekannt wurde das Spielprinzip unter anderem durch das 1998 von Blizzard Entertainment veröffentlichte Spiel *Starcraft* [Ent11].

2.3.1. Typischer Spielablauf

Die Startsituationen in Tower Defense Spielen können stark variieren. Vereinfacht handelt es sich um eine zweidimensionale, rechteckige Fläche, die als *Map* oder Spielfeld bezeichnet wird. Nach einem bestimmten Rhythmus erscheinen Gruppen von gegnerischen Einheiten auf dem Spielfeld. Der Spieler muss mit verschiedenen frei positionierbaren Hindernissen versuchen, die Gegner am Verlassen des Spielfeldes zu hindern. Auf dem Spielfeld befinden sich Markierungen, die angeben auf welchen Seiten des Spielfelds die gegnerischen Einheiten erscheinen und auf welcher Seite sie die Karte wieder verlassen.

Zu Beginn einer Spielrunde kann der Spieler eine begrenzte Anzahl von Hindernissen auf dem Spielfeld platzieren. Hindernisse sind dabei entweder passiv und blockieren den Weg der Gegner, oder aktiv, besitzen also Möglichkeiten die Gegner durch Geschosse, Druckwellen oder andere Waffentypen vom Spielfeld zu entfernen. Anschließend erhält der Spieler für jeden aufgehaltenen Gegner einen bestimmten Kredit an Spielpunkten gutgeschrieben, mit dem sich weitere Hindernisse freischalten und platzieren lassen, um die in jeder Runde stärker werdenden Gegner noch effektiver bekämpfen zu können. Des

Weiteren lassen sich bereits platzierte Hindernisse aufrüsten, so dass diese verbesserte Eigenschaften aufweisen.

Das Spiel gilt als gewonnen, wenn nach einer spieleabhängigen Anzahl von Spielrunden eine vorgegebene Anzahl von Gegnern aufgehalten werden konnten. Der Spieler verliert das Spiel, wenn mehr gegnerische Einheiten das Spielfeld verlassen konnten als die jeweilige Schwierigkeitsstufe erlaubt.

Damit der Spieler dazu angehalten wird, möglichst strategische Verteidigungen aufzubauen, erlauben es die Spielregeln nicht, den Weg der Gegner vollständig zu blockieren.

2.3.2. Beispiele

Desktop Tower Defense

Das flashbasierte Tower-Defense Spiel *Desktop Tower Defense 1.5* [Pre07] ist ein Beispiel für eine Variante mit frei besetzbarem Spielfeld. Die Gegner verwenden dabei einen Wegfindungsalgorithmus, um den kürzesten Weg zwischen den vom Spieler gesetzten Hindernissen zu erkennen. Im Verlauf des Spiels werden die ankommenden Gegner immer mächtiger und zahlreicher, wodurch der Spieler seine Blockaden immer weiter ausbauen muss.

In Spielen dieser Art ist es beliebt, möglichst verschlungene Labyrinthe aufzubauen, in denen die Gegner lange auf dem Spielfeld festgehalten werden. Abbildung 2.1 zeigt eine Spielsituation in der bereits einige Türme, die als graue Rechtecke dargestellt werden, zu einem einfachen Labyrinth zusammengestellt wurden.

Abbildung 2.1.: Desktop Tower Defense 1.5

Abbildung 2.2.: Bloons Tower Defense

Bloons Tower Defense

Bloons Tower Defense [Kai09] besitzt im Gegensatz zu *Desktop Tower Defense* vordefinierte

Spielfelder, auf denen der Weg der Gegner bereits festgelegt wurde. Der Spieler kann die Gegner nur durch am Rand dieses Weges aufgestellte Objekte wie Katapulte oder Schleudern behindern. In der in Abbildung 2.2 abgebildeten Spielsituation stellen die grünen und blauen Ballons die Gegner dar, alle Objekte die sich am Rand der Straße befinden sind vom Spieler positionierte Türme.

Genetic Tower Defense

In einer Arbeit von David Fleming wurde ebenfalls eine Variante eines Tower Defense Spiels entwickelt. Ziel der Arbeit mit dem Titel *Genetic Tower Defense* [Fle10] war ebenfalls der Einsatz von genetischen Strukturen. Allerdings wurde der Begriff des Tower Defense dort anders definiert.

Das Spielfeld besteht bei Fleming aus einem 7x6 Raster, wobei auf jedem Feld eine Spielfigur stehen kann. Das Spiel benötigt einen realen Spieler. Ein weiterer Spieler wird als Gegner durch ein genetisches Programm simuliert. Die Spieler setzen abwechselnd jeweils auf ihrer Seite Kampfeinheiten auf das Spielfeld, die dann in jedem Spielzug ein Feld aufeinander zulaufen. Beim Aufeinandertreffen von Gegnern entscheidet dann der Typ der Einheiten, welche Einheit überlebt. Das genetische Programm übernimmt dabei die Rolle eines zweiten Spielers und versucht, auf die Aktionen des menschlichen Spielers möglichst effektiv zu reagieren. Abbildung 2.3 zeigt einen Screenshot aus diesem Spiel, dessen Prinzip an *Plants vs. Zombies* [Inc11] angelehnt ist.

Abbildung 2.3.: Screenshot aus Genetic Tower Defense von David Fleming

3. Konzept und Umsetzung

3.1. Prinzip

Im Gegensatz zu den aufgezählten bereits existierenden Varianten des Tower Defense, soll im Rahmen dieser Arbeit eine Kombination aus Spiel und genetischem Framework untersucht werden. Dieses soll sich auch als Plattform für weitere Experimente im Bereich der genetischen Programmierung eignen.

Bezogen auf das Spielprinzip bedeutet dies, dass die gegnerischen Einheiten nicht mit Hilfe von Wegfindungsalgorithmen über das Spielfeld wandern, sondern jeweils verschiedene eigenständige Programme darstellen. Jedes einzelne Gegnerprogramm besteht dabei aus einer Reihe von Programmbefehlen, deren Anordnung unter der Anwendung genetischer Algorithmen entstanden sind. Möglich sind dabei Befehle zum Ausrichten (Drehen) und Bewegen (Laufen) der Einheit auf dem Spielfeld. Ein solches Programm kann als Sequenz von Genen angesehen werden.

Die Güte eines Programms ermisst sich daraus, wie gut es die Hindernisse des Spielers umgehen kann.

3.2. Datenstrukturen

3.2.1. Spielfeld

Das Spielfeld ist die Oberfläche, auf der alle spielrelevanten Aktionen ablaufen. Die Größe des Spielfeldes ist variabel, von ihr hängt unter anderem ab, wie viele Hindernisse platziert werden können. Das Spielfeld enthält eine Liste aller Hindernisse, unabhängig davon ob diese passive Hindernisse darstellen oder die Fähigkeit zum Schießen besitzen.

```
data GameMap = GameMap {
     mapWidth   :: Int
    ,mapHeight  :: Int
    ,mapTowers  :: [Tower] }
```

Listing 3.1: Definition des Spielfelds

Der genaue Aufbau der Datenstruktur ist in Listing 3.1 dargestellt, wobei Hindernisse hier allgemein als *Tower* bezeichnet werden.

3.2.2. Einheiten

Jedes genetisch entwickelte Programm repräsentiert im Spiel eine gegnerische Einheit. Die Einheiten bestehen aus den Anweisungen ihres Programms, sowie ihren aktuellen

Parametern wie Position, Ausrichtung und Geschwindigkeit als numerische Werte. Die Implementation der Einheiten in Haskell erfolgt wie in Listing 3.2 gezeigt.

```
data Entity = Entity {
      entGenes :: [Gene]          -- list of instructions
    , angle    :: Float           -- rotation
    , pos      :: (Float, Float)  -- position (x,y)
    , speed    :: Float           -- maximum speed
    , health   :: Int             -- decreases when hit
    , eTrack   :: Tracker }       -- additional statistics
```

Listing 3.2: Definition der Einheiten

Um komplexe Statistiken über die Einheiten erfassen zu können, besitzt jede Einheit ein Tracking-Objekt, welches im Verlaufe des Programms Daten über das Verhalten sammelt, die über die für die Fitness benötigen Informationen hinausgehen. Im übertragenen Sinne kann man sich das Tracking-Objekt wie einen Schrittmesser vorstellen, der am Schuh eines Sportlers befestigt ist und die Anzahl der Schritte zählt, die ausgeführt werden.

In Abbildung 3.1 sind beispielhaft drei Einheiten mit zufälligen Genen aufgeführt.

	t=0	t=1	t=2	t=3	t=4	t=5
Entity 1	WALK 12	TURN 24	WALK 75	WAIT	WALK 13	TURN 129
Entity 2	TURN 77	TURN 24	WALK 55	WALK 90	TURN 21	WAIT
Entity 3	WAIT	TURN 86	TURN 10	WALK 43	TURN 75	WALK 98

Abbildung 3.1.: Beispiel der Gene dreier zufälliger Entities

3.2.3. Hindernisse

Die Hindernisse in Tower Defense Spielen werden als Türme bezeichnet, obwohl sie spieleabhängig auch als andere Gegenstände oder Lebewesen (siehe *Plants vs. Zombies* [Inc11]) dargestellt werden können. Die Datenstruktur der Türme besitzt Ähnlichkeiten mit den Einheiten, da sie ebenfalls eine Reihe von Anweisungen besitzen, die allerdings nicht genetisch veränderbar sind. Die Anweisungen geben dabei den zeitlichen Ablauf ihrer Aktionen an. Die Definition der Datenstruktur ist in Listing 3.3 dargestellt.

```
data Tower = Tower {
    towerActions       :: [TowerAction] -- list of actions
    , towerProjectiles :: [Projectile]  -- list of projectiles
    , towerX           :: Double        -- position on x axis
    , towerY           :: Double        -- position on y axis
    , towerPower       :: Int           -- points that are subtracted
                                        -- from entity health when hit
    , towerRange       :: Double        -- shooting range
    , towerAngle       :: Double }      -- shooting direction
```

Listing 3.3: Definition der Türme

Ein Turm führt in jedem Schritt eine von drei verschiedenen Anweisungen aus. Vergleichbar mit den beweglichen Einheiten besitzen Hindernisse eine neutrale Aktion, bei der sich ihr Status nicht verändert.

In regelmäßigen Abständen wird von aktiven Hindernissen eine Abschuss-Aktion durchgeführt, dabei wird ein Projektil erzeugt welches sich mit konstanter Geschwindigkeit vom Hindernis wegbewegt. Auch Türme besitzen eine Anweisung zur Drehung um einen bestimmten Winkel. Die Rotation des Turmes bestimmt dabei die Schussrichtung.

- SHOOT: Der Turm schießt in die Richtung in die er ausgerichtet ist.
- RELOAD: Der Turm wartet den Zeitschritt ab, ohne eine Aktion durchzuführen (vergleichbar mit der Aktion NOP der gegnerischen Einheiten, um Verwechslungen auszuschließen wurde ein anderer Name gewählt).
- TURN: Der Turm dreht sich um den im Parameter des Gens angegebenen Winkel um die eigene Achse.

3.2.4. Anweisungen

Jedes der genetisch entwickelten Programme besteht aus einer Liste von Anweisungen. Eine Anweisung enthält, wie in Listing 3.4 aufgeführt, einen Befehl an die Einheit, sich auf eine bestimmte Art und Weise zu verhalten.

```
data Command = WALK | TURN | NOP
    deriving (Eq, Ord, Enum, Bounded, Read)

type Gene = (Command, Int)
```

Listing 3.4: Definition der Gene

Jede Anweisung besitzt einen ganzzahligen Parameter, der die Ausführung der Anweisung beeinflusst. Im Folgenden werden die verwendeten Arten von Anweisungen aufgeführt:

WALK

Der Walk-Befehl veranlasst bei seiner Auswertung, dass die Einheit sich um eine festgelegte Distanz auf dem Spielfeld fortbewegt. Die neue Position der Einheit berechnet sich dabei aus der vorhergehenden Position und der Rotation. Bei diesem Schritt wird die neue Position auf mögliche Kollisionen mit dem Spielfeldrand, den Hindernissen und den Geschossen überprüft. Kommt es zu einer Kollision mit einem Geschoss, so gilt die Einheit als getroffen. Einheiten besitzen eine bestimmte Anzahl von Lebenspunkten. Bei jeder Kollision mit einem Geschoss wird dieser Wert verringert. Sind die Lebenspunkte aufgebraucht, gilt die Einheit als besiegt und wird vom Spielfeld entfernt. Bei einer Kollision mit dem Spielfeldrand oder einem anderen Hindernis wird die Einheit nur bis direkt vor das Hindernis bewegt, wodurch sich die Objekte nicht durchdringen können.

Der Befehlsparameter gibt dabei die Weite des Schritts an, den der Walk-Befehl auslöst.

TURN

Mit dem Turn-Befehl kann erreicht werden, dass eine Einheit sich um ihre eigene Achse dreht. Dazu besitzt der Turn-Befehl einen Parameter, der den relativen Winkel angibt, um den sich die Einheit drehen soll. TURN 45 würde die Einheit um 45° im Uhrzeigersinn drehen.

NOP

Die No-Operation Anweisung (NOP) ist ein neutraler Befehl. Die Einheit verbleibt auf ihrer aktuellen Position ohne ihren Status zu ändern. Der Parameter der Anweisung wird dabei ignoriert.

3.2.5. Generation

Eine Generation ist eine Gruppe von Populationen, die im selben Entwicklungszyklus entstanden sind (Listing 3.5).

```
type Generation = [Population]
```

Listing 3.5: Definition der Generation

Die Simulation hat das Ziel, aus der jeweils bestehenden Generation eine Folgegeneration zu erzeugen, indem die beschriebenen genetischen Funktionen ausgeführt werden. Die Hierarchie der Datenstrukturen innerhalb einer Generation wird in Abbildung 3.2 verdeutlicht. Die Buchstaben W,T und N stehen dabei für die Gentypen *walk, turn* und *nop*.

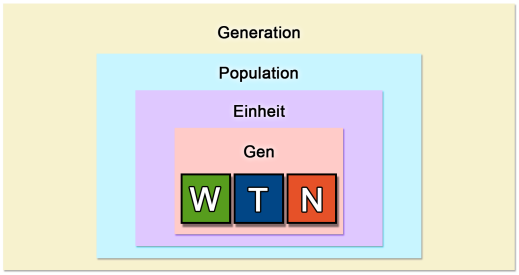

Abbildung 3.2.: Hierarchie der Datenstrukturen

3.2.6. Population

Eine Population ist Teil einer Generation. Sie besteht hauptsächlich aus den in ihr enthaltenen Einheiten. Zusätzlich werden in jeder Population statistische Informationen über die enthaltenen Einheiten abgelegt (Listing 3.6).

```
data Population = Population {
     popEntities:: [Entity]
    ,popStat     :: Statistic }
```

```
data Statistic = Statistic {
     statNumReached   :: Int
    ,statAvgDistance  :: Int
    ,statAvgSteps     :: Int }
```

Listing 3.6: Definition der Population

Für die Berechnung des spezifischen Fitnesswertes einer Population wird hier gespeichert, wie viele der Einheiten das Ziel erreicht haben und welche Zeit sie dafür benötigt haben.

Erreicht eine Einheit das Ziel nicht, wird ermittelt wie nahe sie dem Ziel gekommen ist. Der Durchschnitt dieser Werte über allen Einheiten der Population bildet einen weiteren Parameter, der in der Statistik der Population abgelegt wird.

3.2.7. Konfigurationsobjekt

Die für die Berechnung nötigen Parameter werden in einem eigenen Konfigurationsobjekt zusammengefasst (Listing 3.7). Alle Funktionen mit Zugriff auf dieses Objekt können die für sie relevanten Parameter auslesen und werden auf diese Weise in ihrem Verhalten gesteuert.

```
data Config = Config {
    numGenerations      :: Int -- max number of generations
    ,numPopulations     :: Int -- number of populations per generation
    ,numEntities        :: Int -- number of entities per population
    ,numGenes           :: Int -- number of genes per entity
    ,numStepRes         :: Int -- number of substeps in collision detection
    ,maxSpeed           :: Int -- maximum speed of entities
    ,mutProbability     :: Int -- mutation probability of a single gene
    ,cLatestGeneration  :: IORef (Maybe Generation)
    -- reference to positioning algorithm for testing purposes
    ,cPositioning       :: (Genetics a m) => ([(a,Float)]
                            -> m [(a,Float,Float)])
    -- reference to sampling algorithm for testing purposes
    ,cSampling          :: (Genetics a m) => [(a, Float,Float)]
                            -> StateT [Int] m [a]
}
```

Listing 3.7: Definition des Konfigurationsobjekts

- Die Anzahl der Generationen gibt an, nach wie vielen Evolutionszyklen die Simulation automatisch beendet werden soll, ein vorzeitiges Beenden der Simulation ist zusätzlich über das grafische Interface möglich.
- Die Anzahl der Populationen legt fest, wie viele Gruppen von Einheiten pro Generation existieren. Jede Population enthält dann die eingestellte Anzahl von Einheiten. Dieser Wert hat zusätzlich einen Einfluß auf das Spielverhalten, da die Schwierigkeit des Spiels mit steigender Gegnerzahl ebenfalls ansteigt.

- Wie viele Gene eine Einheit maximal zur Verfügung hat, wird über den Wert *max-Genes* angegeben.
- Die Anzahl der Unterschritte, die bei der Kollisionserkennung und Darstellung angewendet werden sollen, lassen sich ebenso einstellen, wie die allgemeine Mutationswahrscheinlichkeit.
- Für die grafische Ausgabe wird zusätzlich eine Referenz auf die derzeit aktuelle Generation benötigt, die am Ende der Simulation ausgegeben werden soll.
- Da die für die Selektion benötigten Algorithmen als Referenzen angegeben werden, lassen sich zu Testzwecken beliebige Implementationen verwenden.

3.3. Implementierung der genetischen Funktionen

Die hier erzeugten genetischen Programme sind linear, d.h. es existieren weder Sprungbefehle noch Subroutinen.

In der genetischen Programmierung werden komplexe Programme häufig in einer Baumstruktur dargestellt. Die vorliegenden linearen Programme sind in dieser Darstellung also eine sequentielle Aneinanderreihung von Knoten, die einen gemeinsamen Vaterknoten aufweisen. Ein solcher Fall wird als lineare genetische Programmierung bezeichnet [PLM08].

3.3.1. Initialisierung

Zum Beginn der Simulation muss das Spielfeld (Abschnitt 3.2.1) initialisiert sein.

Die genetischen Programme bestehen im ersten Schritt aus zufällig gewählten Anweisungen. Dazu wird eine Kette von zufälligen Genen in einzelne Programme aufgeteilt, die wiederum zu einzelnen Populationen gruppiert werden. Die Länge der Gensequenz eines Programms bestimmt dabei, wie viele Schritte das Programm maximal zum Erreichen des Ziels verwenden darf. Wird die Länge der Sequenzen zu kurz gewählt, kann es den Einheiten unmöglich sein, das Ziel zu erreichen. Sehr lang gewählte Sequenzen haben Einfluss auf die Anfangsperformance, da eine große Menge von Genen ausgewertet werden muss. Wenn die Entwicklung der Einheiten fortschreitet, werden jedoch solche Programme bevorzugt, welche das Ziel mit möglichst wenigen Genen bzw. Anweisungen erreichen, so dass überflüssige Gene ignoriert werden und sich die Auswertungszeit der Programme verkürzt. Erreicht wird dies durch das Einbeziehen der Anzahl der benötigten Gene in die Fitnessberechnung (siehe Abschnitt 3.4.6).

3.3.2. Selektion

In der zweiten Phase werden die Populationen jeder Generation selektiert, so dass nur ausgewählte Populationen in die nächste Generation übernommen werden. Das allgemeine Prinzip der Selektion wurde in Abschnitt 2.2 erläutert. Konkret wurden vier verschiedene Algorithmen ausgewählt, deren Eignung für das gegebene Problem untersucht

werden sollen. Es handelt sich dabei um Algorithmen die sowohl in [ES03] als auch in [PLM08] Erwähnung finden. Aufgrund der großen Zahl verfügbarer Algorithmen musste die Auswahl aus zeitlichen Gründen auf diese Weise eingeschränkt werden.

Kombination der Selektionsalgorithmen

Die Algorithmen Fitness Proportionate Selection und Ranking Selection geben an, wie die Eigenschaften der Populationen die Selektion beeinflussen. Roulette Wheel Selection und Stochastic Universal Sampling bestimmen, wie die Populationen zu einer neuen Generation zusammengesetzt werden können.

Es ist also möglich, die beiden Arten von Algorithmen miteinander zu kombinieren. Da sich die einzelnen Algorithmen problemspezifisch unterschiedlich verhalten können, kann ermittelt werden, welche Kombination für das vorliegende Problem des Tower Defense am besten geeignet ist.

Eine Möglichkeit, dies zu erreichen besteht darin, Messungen mit allen Kombinationen durchzuführen, um so eine möglichst optimale Einstellung zu finden. Dies wurde in Abschnitt 5.1 durchgeführt.

Fitness Proportionate Selection

Das Kriterium zur Auswahl der Populationen ist der spezifische Fitnesswert jeder Population.

Eine neue Generation kann gebildet werden, indem die Wahrscheinlichkeit, dass eine Population in die neue Generation übergeht, direkt proportional zu ihrem Fitnesswert steht. Durch diese *Fitness proportionate* genannte Methode ist sichergestellt, dass gute Populationen deutlich häufiger ausgewählt werden als schlechte Populationen. Eine Population kann auch mehrfach ausgewählt werden. In diesem Fall wird jedesmal eine neue Kopie der Population angelegt. Im Verlauf der Entwicklung nähern sich alle Populationen dem Ziel. Dadurch werden die Abstände der Fitnesswerte immer kleiner, die Populationen werden sich in ihrer Effektivität also ähnlicher. Der Nachteil von *Fitness Proportionate* Selection ist, dass aus Generationen mit sehr ähnlichen Populationen jede Population annähernd gleich wahrscheinlich selektiert wird. Dadurch bietet die erzeugte neue Generation kaum einen Vorteil gegenüber der alten Generation [ES03, S. 59].

Ranking Selection

Das Ranking Selection Verfahren versucht, den erwähnten Nachteil der Fitness Proportionate Selection zu vermeiden.

Es werden nicht direkt die Fitnesswerte der Populationen verglichen, sondern deren Position in einer absteigend sortierten Liste. Die Population mit dem besten Fitnesswert erhält dabei den Rang 1, schwächere Populationen werden der Reihe nach entsprechend ihrer Fitness durchnummeriert. Die Selektion erfolgt nun proportional zu diesen Rangnummern. Auf diese Weise werden auch in Generationen mit stark ähnlichen Populationen die erfolgreichen Populationen deutlich bevorzugt. [ES03, S. 60]

Roulette Wheel Selection

Bei der Roulette Wheel Selection ist der dahinterliegende Ansatz, den einzelnen Populationen einen Sektor eines imaginären Glücksrades wie in Abbildung 3.3 zuzuweisen. Die Größe des Sektors ist dabei abhängig von der Fitness der Population.

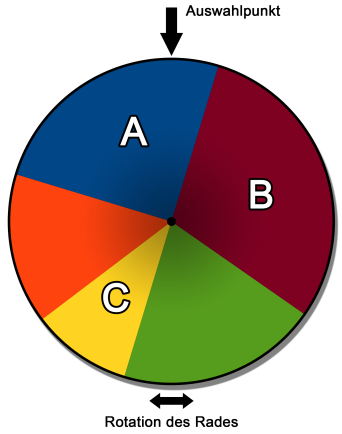

Abbildung 3.3.: Roulette Wheel Selection

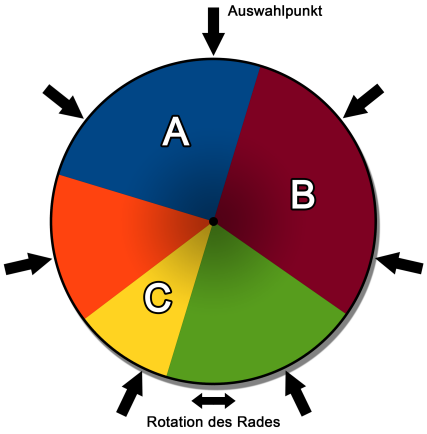

Abbildung 3.4.: SUS (Stochastic Universal Sampling)

Der Umfang des Rades wird als 1 definiert, die Zuteilung der Sektoren erfolgt durch Zuordnung eines Wertes im Bereich [0,1] zu jeder Population. Dieser Wert gibt die Position auf dem Rad an, an dem der zugeordnete Sektor endet. Jeder Wert definiert zugleich das Ende des vorigen und den Beginn des nachfolgenden Sektors. Die Differenz zweier aufeinanderfolgender Werte gibt so die Größe des Sektors an.

Wird nun das Rad gedreht, also eine zufällig ausgewählte Position auf dem Rad ausgewählt, so kann die Population, deren Sektor getroffen wurde, in die neue Generation übergehen.

In Abbildung 3.3 wurde die Population A selektiert, da der Auswahlpunkt sich zwischen den Sektorgrenzen des Sektors A befindet. Zudem ist erkennbar, dass die großen Sektoren A und B beim Drehen des Rades eine deutlich höhere Gewinnchance haben als der kleinere Sektor C.

Da jede Drehung des Rades unabhängig von den vorherigen Drehungen ist, kann keine genaue Aussage über die Verteilung von guten und schlechten Populationen in der produzierten Generation getroffen werden, was zu unerwünschten Ergebnissen führen kann und die Kontrolle über den Algorithmus einschränkt.

Es werden so lange auf diese Weise ausgewählte Populationen zur Generation hinzugefügt, bis die neue Generation wieder die gleiche Anzahl von Populationen enthält wie die vorhergehende.

Welcher Sektor getroffen wurde wird ermittelt, in dem die Sektorgrenzen aufsteigend durchlaufen werden bis eine Sektorgrenze erreicht wird, deren Position hinter der ausgewählten Position liegt [ES03, S. 62].

Stochastic Universal Sampling

Bei dem *Stochastic Universal Sampling* genannten Verfahren kann zur grafischen Verdeutlichung ebenfalls ein Rouletterad verwendet werden. Allerdings werden nun alle Populationen für die nächste Generation mit nur einem einzigen Drehen des Rades bestimmt. Dazu besitzt das Rouletterad (Abbildung 3.4) nicht nur einen Arm der den Gewinner anzeigt, sondern so viele, wie Populationen gezogen werden sollen. Die Arme sind dabei in einem gleichmäßigen Abstand um das Rad verteilt.

Im gezeigten Beispiel werden die durch Sektor A und B repräsentierten Populationen in der neuen Generation zweimal vorkommen, wohingegen die kleineren Sektoren lediglich einmal ausgewählt wurden.

Durch die Gleichverteilung der Arme ist, im Gegensatz zur Roulette Wheel Selection, garantiert, dass die produzierte neue Generation das gewünschte Verhältnis von guten zu schlechten Populationen aufweist. In diesem Fall bedeutet dies, dass die Anzahl der erfolgreichen Populationen in der erzeugten Generation überwiegt.

Ist x also die Anzahl der Populationen, die gezogen werden soll, so kann ein Drehen des Rades simuliert werden durch Erzeugen eines zufälligen Offsets der Arme im Bereich [0, (1/x)] [ES03, S. 62].

3.3.3. Mutation

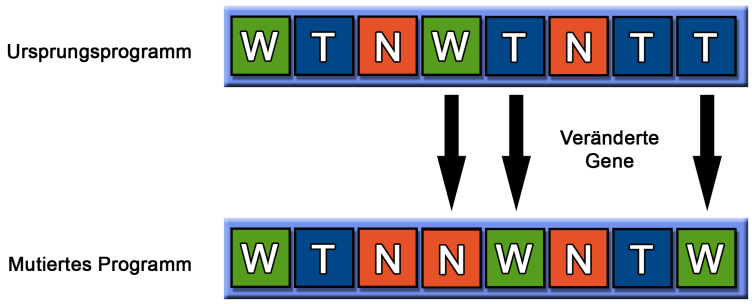

Abbildung 3.5.: Mutation eines Programms

Wie bereits erwähnt beschreibt die Mutation einen Vorgang, der auf jedem Gen, also auf jeder Programmanweisung einzeln ausgeführt wird. In jedem Evolutionsschritt besitzt jedes Gen eine gewisse Wahrscheinlichkeit, dass es in ein Gen eines anderen Typs mutiert. Im vorliegenden Fall kann ein Gen demnach in ein WALK, WAIT oder TURN Gen mutieren.

Ein Zufallswert bestimmt, ob ein Gen mutieren wird. Wenn ein Gen in einen anderen Typ mutiert, kann die Wahrscheinlichkeit für einen bestimmten Typ über eine Gewichtung festgelegt werden. So ist es beispielsweise einstellbar, dass mutierte Gene häufiger

zu einem WALK-Gen mutieren als zu einem NOP-Gen, indem der Mutation zu einem WALK-Gen eine höhere Gewichtung zugeordnet wird.

Zusätzlich wird bei jedem mutierten Gen der jeweilige Parameter ebenfalls mit einem neuen Zufallswert belegt.

3.3.4. Crossover

Beim Crossover werden die Gene von zwei Programmen miteinander gekreuzt. Dazu

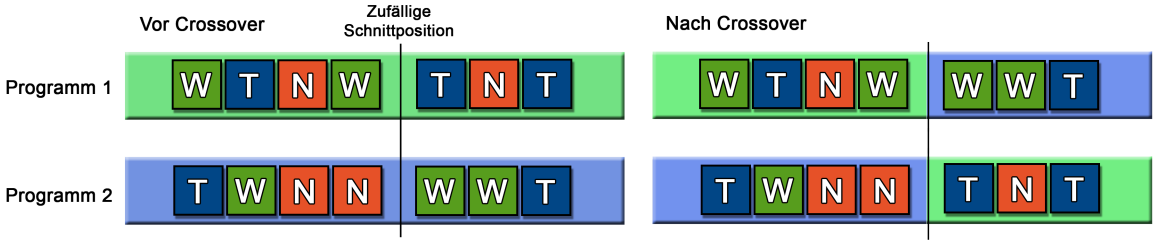

Abbildung 3.6.: Crossover zweier Programme

wird zu jeder Einheit zufällig eine andere Einheit ausgewählt. Die Programme der beiden Einheiten werden dann an einem zufällig gewählten Index in zwei Hälften aufgeteilt.

Nun werden zwei neue Programme erzeugt, indem die jeweiligen Hälften der beiden Programme vertauscht werden (Abbildung 3.6).

Auf diese Weise soll erreicht werden, dass besonders effektive Programmteile zu neuen Programmen zusammengesetzt werden können, die den vorherigen Programmen überlegen sind.

3.4. Implementierungsdetails

3.4.1. Ablauf der Simulation

In diesem Kapitel soll aufgeführt werden, wie eine typische Simulation in Genetic Tower Defense abläuft.

1. Das in Abschnitt 3.2.7 beschriebene Konfigurationsobjekt wird mit den vom Benutzer gewählten Parametern initialisiert.

2. Der Spielplan wird von Datei eingelesen (Abschnitt 3.4.3) oder aus der Eingabe des Benutzers generiert.

3. Alle Programme werden mit zufälligen Gensequenzen initialisiert (Abschnitt 3.3.1).

4. Die Simulation wird der Reihe nach mit allen Populationen durchgeführt. Aus der Leistung der Populationen wird der jeweilige Fitnesswert (Abschnitt 3.4.6) errechnet.

5. Mit einer Kombination der in Abschnitt 3.3.2 beschriebenen Selektionsalgorithmen wird entschieden, welche Populationen in die nächste Generation übergehen werden.

6. Durch Kreuzung der Gene einzelner Einheiten werden im Crossover (Abschnitt 3.3.4) die Programme miteinander vermischt.

7. Mit einer definierten Wahrscheinlichkeit werden einzelne Gene mutiert (Abschnitt 3.3.3).

8. Solange bis die maximal eingestellte Anzahl Generationen erzeugt wurde, oder ein Benutzerabbruch erfolgt, wird der Vorgang ab Schritt 4 mit der zuletzt erzeugten Generation wiederholt.

9. Abschließend kann die letzte erzeugte Generation grafisch ausgegeben werden.

3.4.2. Genetisches Framework

Die für die genetische Entwicklung verantwortlichen Methoden wurde soweit abstrahiert, dass ein eigenständiges genetisches Framework entstanden ist.

Die Klasse Genetics (siehe Listing 3.8) bietet ein abstrahiertes Interface, in dem viele der für genetische Programmierung notwendigen Funktionen bereits definiert sind.

Funktionen wie etwa die Selektionsalgorithmen sind in verschiedenen Varianten vorhanden. Bei der Verwendung der Klasse können diese beliebig kombiniert werden. Wenn dies nicht ausreicht, können auch eigene Implementationen eingebunden werden.

Andere Funktionen, wie etwa die Initialisierung der Populationen sind sehr stark vom jeweiligen Problem abhängig, und müssen daher in der jeweils benötigten Form bereitgestellt werden.

```
class Monad m => Genetics a m where

-- MINIMAL
    initialize :: StateT [Int] m [a]
    mutate :: [a] -> StateT [Int] m [a]
    fitness :: [a] -> StateT [Int] m [(a,Float)]

 -- OPTIONAL
    crossover :: [a] -> StateT [Int] m [a]
    select :: ([(a,Float)] -> m [(a,Float,Float)]) -> ([(a, Float,Float)] ->
        StateT [Int] m [a]) -> [(a,Float)] -> StateT [Int] m [a]
```

Listing 3.8: Ausschnitt der Funktionsdeklarationen der Klasse Genetics

Wurden die Funktionen an das Problem angepasst, kann über die Funktion *evolve* ein vollständiger Generationsschritt durchgeführt werden. Dabei werden die bereits in Abschnitt 2.2 beschriebenen Schritte Evaluierung, Selektion, Crossover und Mutation angewendet.

Jeder Aufruf von *evolve* hat zur Folge, dass sich die genetischen Strukturen um eine Generation weiterentwickeln.

Dieser Vorgang kann beliebig häufig wiederholt werden, bis das Problem mit der gewünschten Genauigkeit gelöst wurde.

In verfügbaren Implementierungen von genetischen Frameworks in Haskell werden ähnliche Ansätze verfolgt. Im Falle von *genprog* [Sna10] liegt der Schwerpunkt beispielsweise auf der Verwendung von in Bäumen strukturierten Daten. Das Paket *hgalib* [Ell08] bietet für verschiedene Arten von genetischen Strukturen bereits vorgefertigte Implementationen an.

Das hier entwickelte Framework ist nicht auf eine bestimmte Datenstruktur angewiesen. Es beschränkt sich darauf, den Ablauf der genetischen Operatoren vorzugeben und bietet die Möglichkeit, problemspezifische Implementationen der Funktionen für Initialisierung, Evaluierung, Selektion, Mutation und Rekombination einzusetzen.

3.4.3. Dateiformat für Spielfelder

Um die Definition von Spielfeldern für die Simulation möglichst simpel zu gestalten, wurde ein einfaches textbasiertes Dateiformat erstellt.

```
400 300
100 150 TS 1 100 180
200 150 TS 1 100 180
300 150 TS 1 100 180
```

Listing 3.9: Beispieldatei eines Spielfelds mit drei Türmen

Listing 3.9 zeigt eine Beispieldatei, in der ein Spielfeld mit drei Türmen definiert wird.

Zwei ganzzahlige Werte in der ersten Zeile jeder Datei dieses Formats geben die Breite und Höhe des Spielplans an. Im obigen Beispiel wird ein Spielplan mit einer Größe von 400x300 Pixeln definiert. Alle folgenden Zeilen enthalten jeweils die Informationen für einzelne Türme:

$$\underbrace{300\ 150}_{\text{a}}\ \underbrace{\text{TSRR}}_{\text{b}}\ \underbrace{1}_{\text{c}}\ \underbrace{100}_{\text{d}}\ \underbrace{180}_{\text{e}}$$

(**a**) Die ersten beiden Werte geben die Koordinaten der Position des Turms an

(**b**) Die danach angegebene Zeichenfolge wird als Kette von Aktionen interpretiert, die später zyklisch durchlaufen werden. Jeder Buchstabe steht für eine Aktion, die der Turm in einem Zeitschritt ausführen kann (siehe 3.2.3).

(**c**) Die Stärke der Geschosse, die dieser Turm verschießen kann in Punkten

(**d**) Die Reichweite der Geschosse in Pixeln

(**e**) Ausrichtung bzw. Drehung des Turmes in Grad

In der Simulation werden die angegeben Aktionen der Reihe nach ausgeführt. Wenn das Ende der Zeichenkette erreicht wurde, wird wieder mit der ersten Aktion begonnen.

3.4.4. Kollisionserkennung

Die Programme werden in der Spielwelt als Türme und gegnerische Einheiten dargestellt. Da die Objekte als Festkörper betrachtet werden sollen die sich nicht durchdringen können, bestimmt ihre Größe, wie weit sie sich gegenseitig nähern können.

Die Kollisionserkennung sorgt dafür, dass Objekte sich nicht ineinander bewegen können und verhindert somit eine Durchdringung der Objekte [MW88].

Im vorliegenden Fall kann zwischen drei unterschiedlichen Arten von Kollisionen unterschieden werden:

- Bei der Kollision einer Einheit mit dem Spielfeldrand wird sichergestellt, dass die Einheit das Spielfeld nicht verlassen kann. Läuft eine Einheit schräg gegen den Rand, so wird sie nicht abrupt stehenbleiben, sondern wird an der Wand entlang gleiten.

- Kollidiert eine Einheit mit einem Hindernis, so wird sie vor dem Hindernis stehen bleiben.

- Die Kollision mit einem Geschoss hat keinen Einfluss auf die Bewegung der Einheit. Allerdings wird der Einheit bei einer Kollision durch Abzug von Punkten Schaden zugefügt, der unter Umständen dafür sorgt, dass die Einheit vom Spielfeld entfernt wird.

Werden die Einheiten des Spiels als Punkte in einem zweidimensionalen Raum betrachtet, so lassen sich die Bewegungen aller Objekte in einem Zeitschritt als Vektoren darstellen. Eine Kollision kann nur dann erfolgen, wenn zwei Vektoren sich kreuzen. Anhand der Geschwindigkeit mit der sich die Objekte bewegen lässt sich nun bestimmen, ob beide Objekte zur gleichen Zeit am Schnittpunkt der Vektoren ankommen. In diesem Fall hat eine Kollision stattgefunden.

Die Betrachtung der Objekte als Punkte auf einer Ebene reicht allerdings nicht mehr aus, wenn die Objekte als Flächen dargestellt werden sollen.

Selbst bei einer Abstraktion der Objekte zu Kreisen mit definierten Radien, ist die Bestimmung einer Kollision nicht mehr ohne zusätzlichen Aufwand möglich. Kollisionen können nun auch dann auftreten, wenn die Vektoren der Einheiten sich nicht schneiden. In diesem Fall kann über die Radien der Objekte ermittelt werden, ob diese kollidieren.

Die spätere grafische Animation der Simulation wird aus Einzelbildern bestehen, die mit konstanter Geschwindigkeit abgespielt werden.

Für die Kollisionserkennung ist es also ausreichend, wenn für jedes dieser Einzelbilder, die jeweils Momentaufnahmen der Simulation widerspiegeln, sichergestellt ist, dass sich die Objekte nicht durchdringen. Um dies zu erreichen werden die Bewegungen der Objekte in Unterschritte aufgeteilt, die jeweils einem Einzelbild der späteren Animation entsprechen. In jeder dieser Momentaufnahmen wird nun geprüft, ob sich Objekte überschneiden. Ist dies der Fall, so wird die Anzeige korrigiert.

Für die Erkennung einer Kollision wird die Position jedes Objektes auf dem Spielfeld mit den Positionen aller anderen Objekte verglichen. Da die Ausmaße der Objekte als Kreise betrachtet werden, liegt eine Kollision immer dann vor, wenn sich die Kreise zweier Objekte schneiden. Das ist genau dann der Fall, wenn die Summe der beiden Radien größer ist als der Abstand der Mittelpunkte der Kreise.

Der Abstand der zwei Mittelpunkte kann über die Koordinaten der Punkte mit Hilfe des Satzes von Pythagoras berechnet werden:

$$d_{AB} = \sqrt{(x_1 - x_2) + (y_1 - y_2)}$$

x_i und y_i beschreiben die entsprechende Position des Punktes i auf der x- bzw. y-Achse. Eine Kollision liegt immer dann vor, wenn der boolsche Ausdruck (3.1) wahr ist.

$$doesCollide = d_{AB} < r_A + r_B \tag{3.1}$$

Hierbei beschreiben r_a und r_b die jeweiligen Radien der beiden Objekte A und B.

Da jeweils alle Objekte mit allen anderen Objekten verglichen werden, beträgt die Laufzeit hier $O(n^2)$, wobei n die Anzahl der Objekte auf dem Spielfeld angibt.

Auf eine weitere Optimierung der Kollisionserkennung wurde aus Zeitgründen verzichtet.

3.4.5. Auswertung der Gene

In jedem Zeitschritt werden die Gene aller Einheiten nacheinander ausgewertet. Im Zeitschritt n werden alle Gene interpretiert, die in den Programmen der Einheiten an Index n der Anweisungsliste stehen. Die im Konfigurationsobjekt (siehe Abschnitt 3.2.7) eingestellte Anzahl der Unterschritte bestimmt dabei, in wie viele Teilschritte jedes Gen bei der Evaluierung zerlegt werden muss. Dazu wird der ganzzahlige Parameter eines Gens durch die Anzahl der Unterschritte dividiert.

$$\text{Parameter der Teilschritte} = \frac{\text{Parameter des Genes}}{\text{Anzahl der Unterschritte}} \tag{3.2}$$

Eine Aufteilung in 20 Unterschritte hätte beispielsweise zur Folge, dass ein WALK-Gen mit einer Schrittweite von insgesamt 40 Längeneinheiten in Unterschritte aufgeteilt wird, deren Länge jeweils 2 Längeneinheiten entspricht. Nach jedem der Teilschritte werden die in Abschnitt 3.4.4 genannten Kollisionstests durchgeführt.

Auf diese Weise ist in jedem Teilschritt sichergestellt, dass auftretende Kollisionen erkannt werden können.

3.4.6. Fitnessberechnung

Nachdem die Programme einer Population evaluiert worden sind, dass heißt wenn festgestellt wurde, wie viele Programme einer Population das Spielfeld überqueren konnten, geben die gesammelten Daten Aufschluss über die Performance der Population.

Aus den ermittelten Daten kann nun ein einzelner Skalar berechnet werden, der einen Vergleich der Populationen zulässt. Dieser Wert wird als spezifischer Fitnesswert der Population bezeichnet.

Die zentrale Frage an dieser Stelle ist: Was ist eine erfolgreiche Population? Das Ziel aller Einheiten ist, das Spielfeld zu überqueren. Je mehr Einheiten einer Population dieses Ziel erreichen, desto erfolgreicher ist die Population als Ganzes. Zu Beginn der Simulation werden nur sehr wenige Einheiten das Ziel bereits erreichen können, da die Gene noch zu einem Großteil zufällig verteilt sind.

Mit dem vorgestellten Kriterium lässt sich keine Aussage über den Erfolg einer Population machen, deren Einheiten das Spielfeld nicht überqueren konnten.

Um auch solche Populationen bewerten zu können, wird betrachtet, wie nah die Einheiten dem als Ziel definierten Spielfeldrand gekommen sind. Ein Wert im Bereich $[0, 1]$ gibt dabei die Distanz an.

Einheiten, die das Spielfeld vollständig überquert haben, werden mit einer 1 bewertet. Alle anderen Einheiten bekommen einen Wert kleiner 1. Je näher dieser Wert bei 1 liegt, desto näher ist die Einheit dem zu erreichenden Spielfeldrand gekommen.

Ein weiterer Aspekt, der in die Berechnung des Fitnesswertes mit einfließt, ist die Zeit, die die Einheiten benötigen, um das Spielfeld zu überqueren.

Als Messgröße für die Zeit wird dabei ermittelt, wie viele Einzelschritte die Einheit für das Überqueren des Ziels benötigt hat. Eine Einheit, die nur wenige Schritte zum Erreichen des Ziels braucht, wird höher bewertet, als eine Einheit, die viele Schritte benutzt hat.

Da der Fitnesswert der Populationen aus mehreren Faktoren (Entfernung und Zeit) besteht, handelt es sich um sogenanntes *multi-objective genetic programming* [PLM08, S. 75,ff].

Ein in [Koz92] beschriebener Weg, die einzelnen Faktoren zu einem Fitnesswert f_i zu kombinieren, besteht darin, sie in einer linearen Funktion zusammenzufassen.

$$f_i = \alpha \cdot x_1 + \beta \cdot x_2 + \dots$$

Setzt man für x_1 und x_2 nun die genannten Faktoren für Zielabstand und Geschwindigkeit ein, erhält man die in (3.3) gezeigte Formel.

$$f_i = \alpha \cdot \frac{p_y}{m_{length}} + \beta \cdot \frac{1}{\text{stepcount}} \tag{3.3}$$

p_y ist hierbei die Entfernung, die eine Einheit in Richtung des Ziels zurückgelegt hat. m_{length} beschreibt die Entfernung des Ziels vom Startpunkt aus. Je größer m_{length}, desto näher ist die Einheit dem Ziel gekommen.

Die Faktoren α und β sind dabei frei wählbar. Mit ihnen lässt sich eine Gewichtung der Ziele einstellen. Über eine Gleichgewichtung der Faktoren kann erreicht werden, dass keines der Ziele gegenüber dem anderen bevorzugt wird.

Sobald die Simulation einen Zustand erreicht hat, in dem alle Einheiten das Ziel erreichen können, gilt für alle Einheiten $p_y = m_{length}$. Bei der weiteren Entwicklung ist dieser Faktor also bei allen Einheiten konstant und die Verkürzung des zurückgelegten Weges ist das einzige verbleibende Ziel.

Aus diesem Grund ist der Effekt, den die Gewichtungsfaktoren auf das Resultat der Simulation haben, gering.

4. Darstellung

Durch die Trennung von Simulation und Ausgabe ist *Genetic Tower Defense* nicht an eine bestimmte Art der Darstellung der Ergebnisse gebunden. Im Laufe der Entwicklung des Kernprogramms wurden drei verschiedene Ausgabeverfahren implementiert, die sich sowohl in ihrer Komplexität, als auch in der verwendeten Technologie unterscheiden.

4.1. Textbasierte Konsolenausgabe

Über die Konsole des jeweiligen Betriebssystems können Programme textbasierte Informationen für den Anwender ausgeben.

Für *Genetic Tower Defense* ist es in diesem Fall sinnvoll, mit Hilfe einer regelmäßigen Statusausgabe auf der Konsole über die aktuellen Populationen und deren Eigenschaften zu informieren.

Eine Ausgabe sämtlicher Gene einer oder mehrerer Populationen in der Konsole ist zwar möglich, ist aber aufgrund der entstehenden großen Datenmenge nur in Einzelfällen und zur Fehlersuche sinnvoll.

Stattdessen werden in jedem Entwicklungsschritt, also in jeder Generation, für jede Population nur mathematische Mittelwerte der Parameter ausgegeben. Relevant sind hier besonders solche Werte, die bei der Fitnessberechnung einbezogen werden. Mit ihnen lässt sich die Performance der einzelnen Populationen im Verhältnis zu anderen Populationen der selben Generation abschätzen. Ein Beispiel einer solchen Ausgabe erfolgt in Listing 4.1.

Pro Zeile werden die Daten aller Populationen einer Generation ausgegeben. Zusätzlich erfolgt die Ausgabe des Fitnesswertes der erfolgreichsten Population. So lässt sich die Performance der Generation mit den vorangegangenen und folgenden Generationen vergleichen.

```
111: 9(1)(11) 9(1)(11) 10(0)(11) 10(0)(12) - Best fitness value: 18.569294
112: 10(0)(11) 9(11)(10) 10(0)(12) 10(0)(12) - Best fitness value: 18.830313
113: 10(0)(11) 9(19)(10) 10(0)(12) 10(0)(12) - Best fitness value: 18.726446
114: 9(19)(9) 10(0)(11) 10(0)(11) 10(0)(12) - Best fitness value: 19.018616
```

Listing 4.1: Ausgabe auf der Konsole

Die Ausgabe der Populationsdaten erfolgt dabei in Wert-Tripeln. Das erste Tripel in Listing 4.1, 9(1)(11) ist folgendermaßen zu interpretieren:

9 Anzahl der Einheiten, die das Spielfeld erfolgreich überqueren konnten

(1) Summe der Abstände aller Einheiten zum Ziel

(11) Anzahl der Gene, die durchschnittlich pro Einheit ausgewertet werden mussten

Eine leicht modifizierte Variante dieser Ausgabe wird in eine Datei geschrieben.

4.2. Textgrafiken

An Hand der auf der Konsole ausgegebenen Daten in Form von Zahlenreihen lassen sich zwar bereits viele Informationen ablesen, die genauen Abläufe der Programme, und insbesondere der genaue Weg, den die Einheiten nehmen um das Ziel zu erreichen, lässt sich auf diese Weise allerdings noch nicht erkennen.

Es ist nicht möglich, direkt auf der Konsole Grafiken auszugeben. Allerdings lassen sich in der Ausgabe bestimmte Buchstaben und Symbole auf eine Weise anordnen, dass sie eine Spielsituation von *Genetic Tower Defense* nachbildet.

```
################################################################################
#       E                                                                      #
#                                                                              #
#                                  E                T                          #
#                                                                              #
#                                                                              #
#                                                                              #
#                                                   E                          #
#                                                                              #
#                                                                              #
#                                                                              #
#                                                                  T           #
#                T                                                             #
#                                       E                                      #
#                                                                              #
#                                                                              #
#                                                                              #
#                                                   E                          #
#                                                                              #
#                                                                              #
#                                  T                                           #
#                E                                                             #
#                                                                              #
################################################################################
```

Listing 4.2: Pseudografische Ausgabe auf der Konsole

Wie in Listing 4.2 erkennbar, wird das Spielfeld durch einen umgebenden Rahmen markiert. Die Einheiten auf dem Spielfeld werden durch den Buchstaben 'E' an ihrer jeweiligen Position gekennzeichnet. Türme werden durch ein 'T' dargestellt. Die Darstellung von Schüssen ist nicht umgesetzt, da nur die grundlegende Funktionalität getestet werden soll. Eine umfangreichere Darstellung wird in Abschnitt 4.3 vorgestellt.

Wenn das Konsolenfenster die selbe Größe hat wie das Spielfeld, kann der Eindruck von animierten Einheiten erzeugt werden, in dem die Simulationsschritte in schneller Folge hintereinander ausgegeben werden. Durch das schnell wechselnde Bild wird der Eindruck von Bewegung erzeugt und der Weg der Einheiten lässt sich visuell nachvollziehen.

4.3. Grafisches Interface

4.3.1. GTK+

GTK+ ist ein Toolkit, mit dem sich plattformübergreifend grafische Benutzeroberflächen erzeugen lassen. Dazu wird dem Anwender eine API zur Verfügung gestellt, die sich in

einer Vielzahl von Programmiersprachen ansprechen lässt. Haskell zählt, neben C, C++, Python und einigen anderen, zu den unterstützten Programmiersprachen.

4.3.2. Glade

Das Programm Glade ist ein Editor, mit dem sich auf GTK+ basierende grafische Benutzeroberflächen erstellen lassen. Als grafischer Editor bietet Glade vordefinierte Bedienelemente, wie z.B. Textfelder, Eingabefelder und ähnliches, die per Drag&Drop zu komplexen Interfaces zusammengesetzt werden können. Die eigentliche Funktionalität lässt sich nicht mit Glade erstellen. Sie wurde in diesem Fall durch Benutzung der dafür vorgesehenen Pakete in Haskell implementiert.

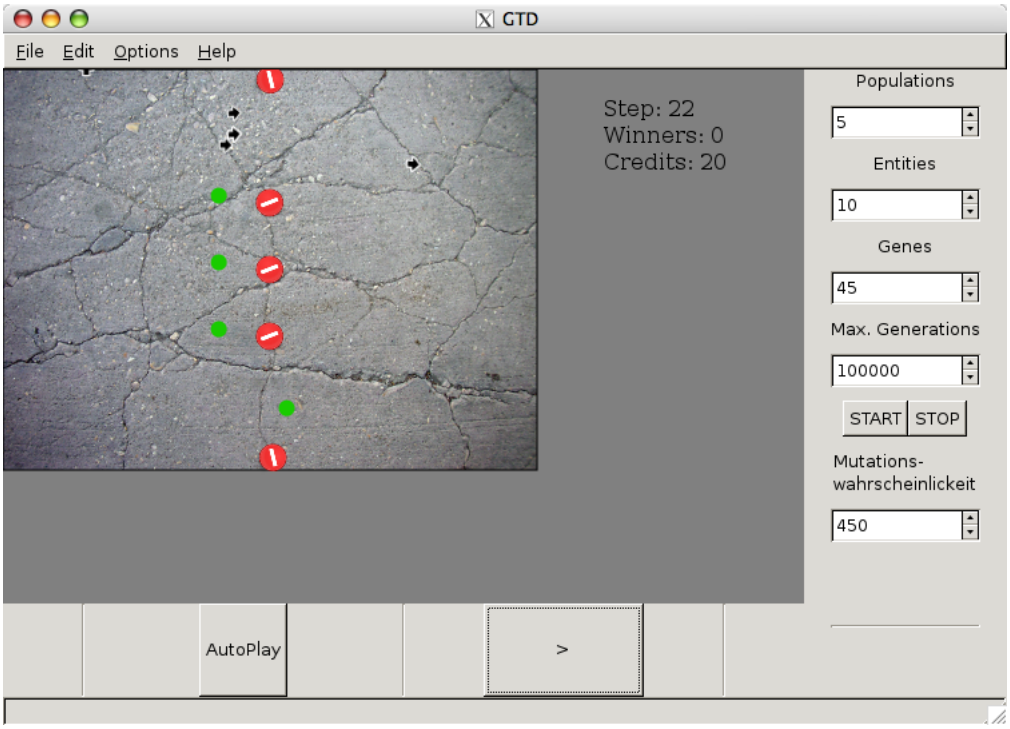

Abbildung 4.1.: Ansicht der Benutzeroberfläche

Die mit Glade erstellte Benutzeroberfläche lässt sich in einer XML-Datei abspeichern, die bei der Verwendung in Genetic Tower Defense geladen werden kann.

Abbildung 4.1 zeigt die Benutzeroberfläche von Genetic Tower Defense mit der Darstellung des Spielfeldes und den Einstellungsmöglichkeiten.

Auf der rechten Seite der Oberfläche können die numerischen Parameter des Konfigurationsobjekts eingestellt werden.

Auf der Darstellung des Spielfeldes lassen sich, durch einen einfachen Klick auf die gewünschte Stelle, Hindernisse an beliebige Positionen setzen. Die Hindernisse werden dabei als rote Kreise dargestellt. Anschließend erlaubt der Button *Start*, die Simula-

tion mit den eingestellten Parametern zu beginnen. Während der Berechnung wird der aktuelle Status auf der Konsole ausgegeben, bis die eingestellte maximale Anzahl von Generationen erreicht wurde, oder der Benutzer die Berechnung mit Hilfe des *Stop*-Buttons beendet.

Wenn die Berechnung abgeschlossen ist, kann über das Menü im unteren Bereich der Oberfläche mit dem Pfeil-Button die beste Population der letzten Generation schrittweise angezeigt werden. Dabei werden die Bewegungen der Einheiten und Türme animiert auf dem Spielfeld ausgegeben. Die Einheiten werden durch bewegte Pfeile dargestellt, deren Ausrichtung die Blickrichtung der Einheiten widerspiegelt. Alternativ erlaubt der Button *AutoPlay* ein automatisches Abspielen aller Schritte der Population.

Eine Anzeige rechts des Spielfeldes zeigt an, welcher Schritt der Simulation gerade angezeigt wird. Ein Schritt entspricht dabei einem Zeitschritt von Genen in der Simulation.

5. Messungen

Die genetische Programmierung bietet eine Vielzahl von verschiedenen Techniken und Algorithmen. Dies macht ihre Anwendung sehr flexibel. Es bedeutet aber auch, dass für jedes spezifische Problem unterschiedliche Kombinationen von Techniken mehr oder weniger effektiv sein können.

5.1. Selektion

Als Beispiel sei hier der Vorgang der Selektion genannt. Allein die in Abschnitt 3.3.2 vorgestellten Ansätze erlauben es, vier verschiedene Arten von Selektion durchzuführen, indem die Algorithmen unterschiedlich kombiniert werden. Um die Selektionsalgorithmen und ihre Auswirkung auf die Effektivität des Programms vergleichen zu können, wurden verschiedene Messungen vorgenommen.

Damit die Vergleichbarkeit der Ergebnisse gewährleistet ist, wurden die Algorithmen jeweils mit identischen Parametern und Eingabewerten mit Hilfe eines zu diesem Zweck geschriebenen Hilfsprogramms mehrfach auf einem Testrechner ausgeführt. Die Daten der einzelnen Testläufe wurden durch Umleitung der in Abschnitt 4.1 beschriebenen textbasierten Ausgaben in einzelnen Logdateien gespeichert.

Bei der späteren Auswertung der Daten wurden jeweils die Mittelwerte der einzelnen Testläufe gebildet.

Folgende Parameter wurden für den Vergleich der Selektionsalgorithmen gewählt:

Anzahl der Tests pro Algorithmus:	500
Anzahl der Generationen pro Test:	600
Anzahl Populationen pro Generation:	10
Anzahl Einheiten pro Population:	10
Maximale Gene pro Einheit:	30
Maximale Geschwindigkeit der Einheiten:	60 Pixel pro Zeitschritt
Auflösung für Kollisionstests:	10 Unterschritte pro Gen

Die grafische Darstellung der Fitnesswerte der einzelnen Generationen zeigt Abbildung 5.1.

Die Messungen haben ergeben, dass die Kombination aus Stochastic Universal Sampling und Ranking Selection von allen getesteten Kombinationen zu den besten Ergebnissen führt. Im Vergleich zu den anderen drei untersuchten Einstellungen steigt der Fitnesswert bereits nach wenigen Generationen sehr stark an.

Da nur die Kombination aus Roulette Wheel und Ranking Selection ähnlich schnell ansteigt, ist diese Wirkung auf den Einsatz der Ranking Selection zurückzuführen. Dies

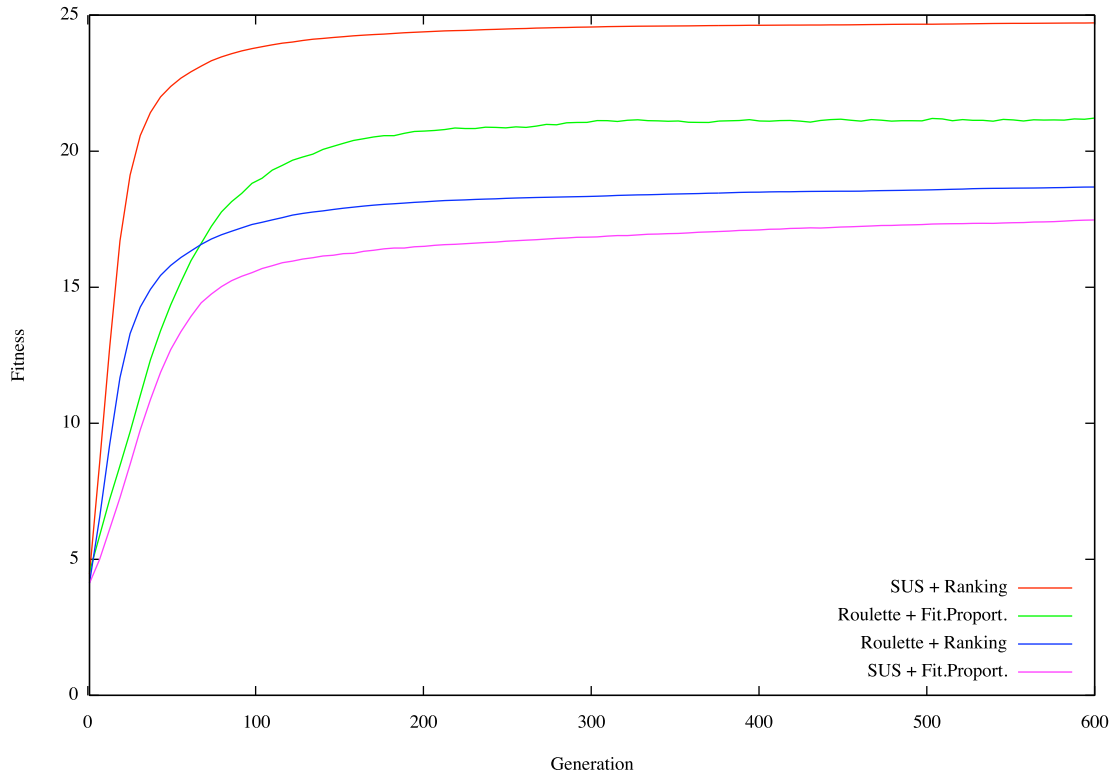

Abbildung 5.1.: Durchschnittliche Fitness der Generationen für vier Algorithmenkombinationen

deckt sich mit dem in [ES03, S. 59] und [Bak87] beschriebenen Verhalten. Grund dafür ist der geringere Evolutionsdruck der Fitness Proportionate Selection, da dort im Vergleich zur Ranking Selection nur sehr große Unterschiede zwischen den Populationen zu einer gerichteten Auswahl führen.

Bei der Roulette Wheel Selection ist der Unterschied zwischen der Kombination mit Ranking oder Fitness Proportionate Selection weniger deutlich. Die Ergebnisse liegen in beiden Fällen im Mittelfeld.

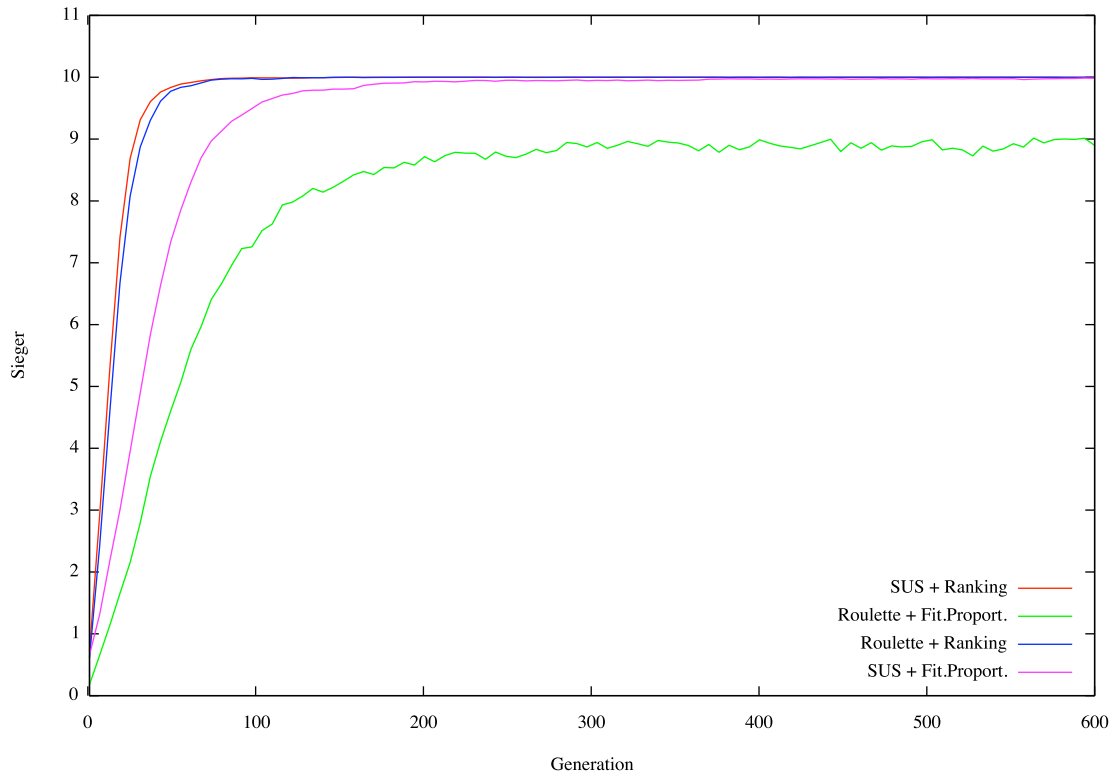

Abbildung 5.2.: Anzahl der Gewinner pro Generation für vier Algorithmenkombinationen

Abbildung 5.2 zeigt das Resultat der Messungen, wenn statt des Fitnesswertes nur die Anzahl von Einheiten betrachtet wird, die das Spielfeld überqueren konnten.

Hier steigt die Fitness der beiden Kombinationen mit Ranking Selection schneller als die der beiden anderen Varianten. Auffällig ist das Verhalten bei der Kombination aus Roulette Wheel und Fitness Proportionate Selection. Der geringe Evolutionsdruck der Fitness Proportionate Selection, verbunden mit den ungünstigen Eigenschaften des Roulette Wheel sorgen dafür, dass die Anzahl der siegreichen Einheiten stark schwankt.

5.2. Mutationswahrscheinlichkeit

Die in Abschnitt 5.1 durchgeführten Messungen der Selektionsalgorithmen wurden mit einer fest eingestellten Mutationswahrscheinlichkeit von 1 : 100 durchgeführt. Nun soll untersucht werden, ob die eingestellte Wahrscheinlichkeit einen Einfluss auf die Performance der Algorithmen hat. Dazu werden alle Kombinationen der vorgestellten Algorithmen mit Mutatationswahrscheinlichkeiten im Bereich 1 : 25 (häufige Mutation) bis 1 : 100000 (seltene Mutation) getestet. Zusätzlich wird untersucht, welche Folgen eine Deaktivierung der Mutation hat.

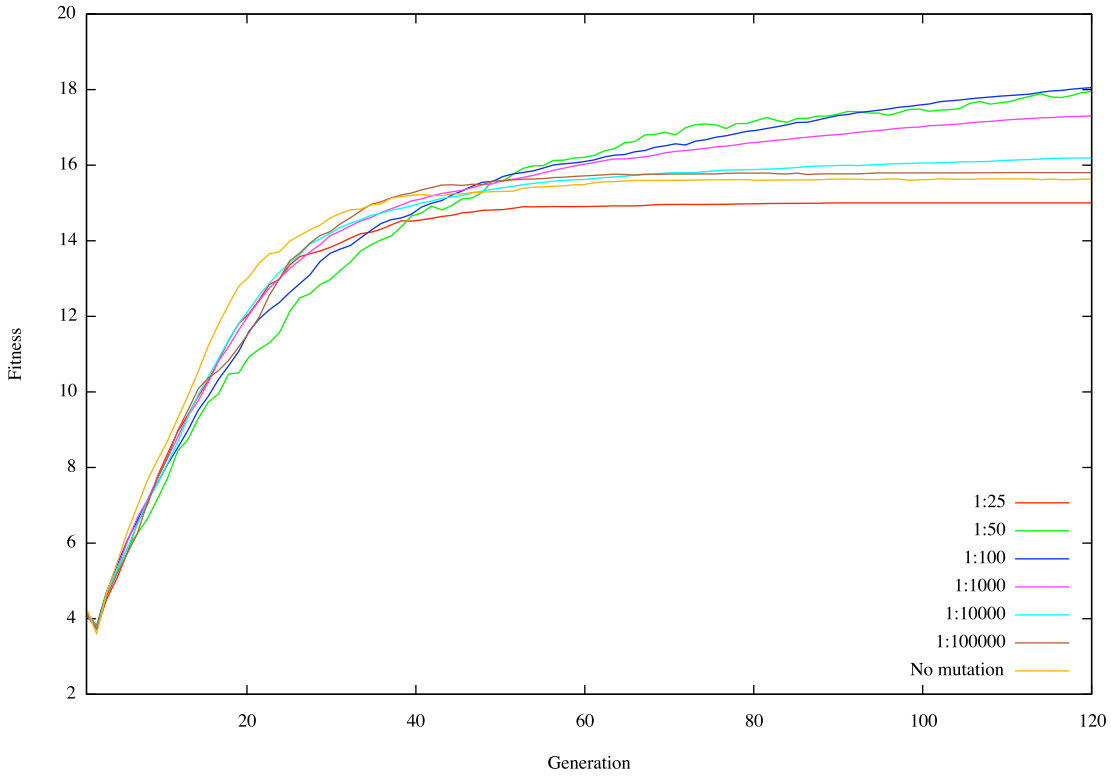

Abbildung 5.3.: SUS und Ranking Selection bei veränderten Mutationswahrscheinlichkeiten

Die Auswirkung der Mutationswahrscheinlichkeit auf den Fitnesswert bei Verwendung von SUS (Stochastic Universal Sampling) und Ranking Selection zeigt Abbildung 5.3.

Wird bei der Simulation auf die Mutation der Gene verzichtet, so steigt die Fitness der Populationen zu Beginn vergleichsweise stärker an. Nach einigen Generationen stagniert die Entwicklung allerdings auf einem fast konstanten Fitnesswert, der unterhalb der Werte liegt, die unter Einsatz der Mutation möglich sind.

Sobald Gene mit einer geringen Wahrscheinlichkeit mutieren können, steigt der Fitnesswert auch in späteren Generationen noch leicht an. Je höher die Mutationswahrschein-

lichkeit gewählt wird, desto höher steigt die Fitness in späten Generationen.

Dieser Effekt kann bis zu einer Mutationswahrscheinlichkeit von 1 : 100 beobachtet werden. Ab diesem Wert beginnt die Fitness der Populationen stark zu schwanken. Wird die Wahrscheinlichkeit für Mutationen weiter erhöht, so sinkt die Fitness auf Werte ab, die sogar unter den Ergebnissen liegen, die bei einem Verzicht auf Mutationen erreicht werden.

Ohne Mutation werden die Programme ausschließlich durch Rekombination bereits bestehender Programme neu geordnet. Durch die Mutation werden neue Programmstrukturen geschaffen, die den vorhandenen Programmen überlegen sein können.

Allerdings bewirkt eine zu hohe Rate der Mutationen eine Verschlechterung der Programme, da durch das häufige zufällige Ändern der Gene keine zielgerichtete Entwicklung mehr möglich ist.

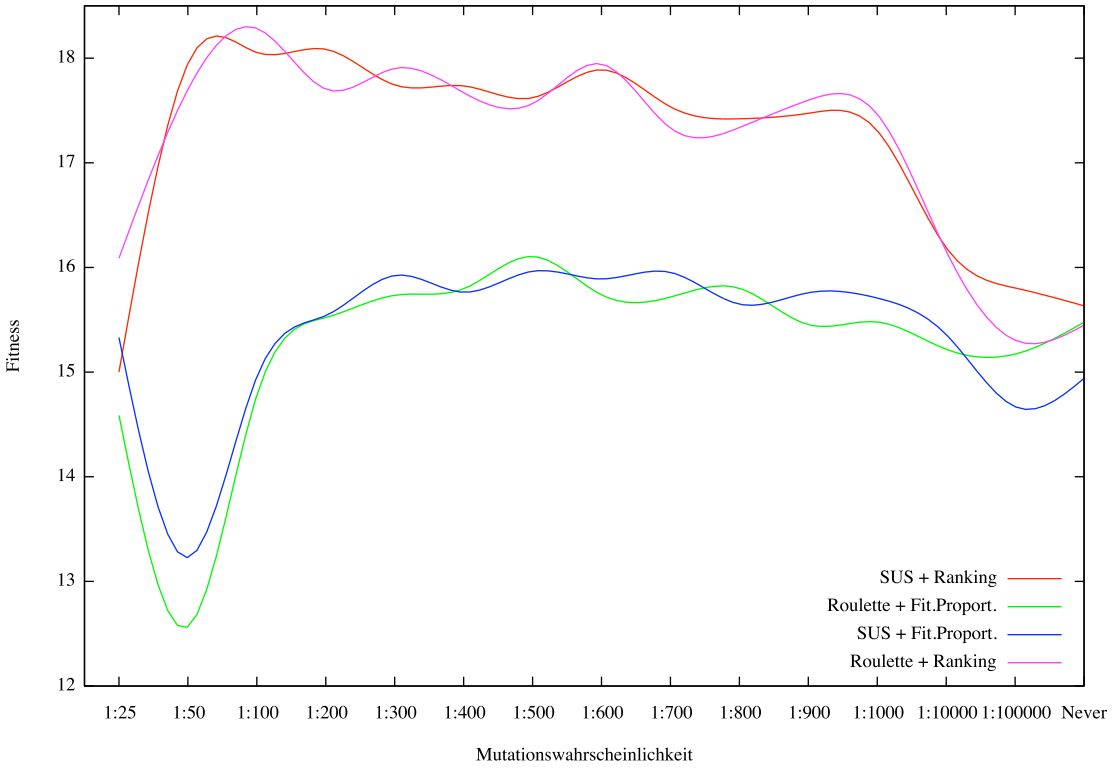

Abbildung 5.4.: Selektionsalgorithmen bei verschiedenen Mutationswahrscheinlichkeiten nach jeweils 120 Generationen

Werden die verschiedenen Mutationswahrscheinlichkeiten bei den vier vorgestellten Selektionsalgorithmen (Abbildung 5.4) verglichen, so wird deutlich, dass die Verwendung der Ranking Selection zu höheren Fitnesswerten führt als der Einsatz von Fitness Proportionate Selection.

Stochastic Universal Selection oder Roulette Wheel Selection haben dagegen keinen

erkennbaren Einfluss auf das Verhalten bei unterschiedlichen Mutationswahrscheinlichkeiten. Die dargestellten Kurven liegen jeweils sehr nah beieinander.

Weiterhin ist ersichtlich, dass ein Verzicht auf Mutation (in Abbildung 5.4 als 'Never' auf der x-Achse eingetragen) bei der Entwicklung in allen vier Kombinationen zu schlechteren Werten führt. Besonders der Vorteil der Ranking Selection gegenüber der SUS geht bei der Deaktivierung des Mutationsvorgangs verloren.

6. Fazit

Die genetische Programmierung stellt eine Möglichkeit dar, den Gegnern in Computerspielen eine künstliche Intelligenz zu verleihen.

Es hat sich gezeigt, dass nicht alle existierenden Ansätze für genetisch basierte Algorithmen für die Simulation von Gegnern in Tower Defense Spielen geeignet sind. Durch umfangreiche Untersuchungen ist es aber gelungen, unter einer Auswahl von untersuchten Algorithmen die effizientesten Kombinationen zu ermitteln. Dabei hat sich besonders die Kombination von *Stochastic Universal Sampling* mit *Ranking Selection* als vorteilhaft erwiesen.

Die gewonnenen Erkenntnisse über die Kombination von Tower Defense Spielen und sich genetisch entwickelnden Einheiten können dazu verwendet werden, neuartige Spielkonzepte zu entwickeln. In einem weiteren Schritt können diese Spiele beispielsweise über eine Online-Spieleplattform einer größeren Zahl von Spielern vorgestellt werden, um den Spielwert dieses Konzepts weiter untersuchen zu können.

Das im Rahmen dieser Arbeit entwickelte Framework kann als Grundlage zur Entwicklung von weiteren Programmen dienen, in denen die genetische Programmierung zur Problemlösung eingesetzt werden soll.

A. Abbildungsverzeichnis

B. Literaturverzeichnis

[Bak87] James E. Baker. Reducing bias and inefficiency in the selection algorithm. In John J. Grefenstette, editor, *Genetic Algorithms and their Applications (ICGA'87)*, pages 14–21, Hillsdale, New Jersey, 1987. Lawrence Erlbaum Associates.

[Dar59] C. Darwin. *On the origin of species*. John Murray, London, 1859.

[Ell08] Kevin Ellis. Haskell genetic algorithm library, 2008. `http://hackage.haskell.org/package/hgalib`.

[Ent11] Blizzard Entertainment. Starcraft - a real-time strategy game, January 2011. `http://us.blizzard.com/en-us/games/sc/`.

[ES03] A. E. Eiben and J. E. Smith. *Introduction to evolutionary computation*. Natural computing series. Springer-Verlag, 2003.

[Fle10] David Fleming. Genetic tower defense. Master's thesis, University of Bolton, 2010.

[Hoa62] C. A. R. Hoare. Quicksort. *The Computer Journal*, 5(4):10–15, 1962.

[Hut07] Graham Hutton. *Programming in Haskell*. Cambridge Univ. Press, NY/Cambridge, UK, 2007.

[Inc11] PopCap Games Inc. Plants vs. zombies - a tower defense game, January 2011. `http://www.popcap.com/extras/pvz/`.

[Kai09] Kaiparasoft. Bloons tower defense 4 - online game, 2009. `http://ninjakiwi.com/Games/Tower-Defense/Play/Bloons-Tower-Defense-4.html`.

[Koz92] John R. Koza. *Genetic Programming: On the Programming of Computers by Means of Natural Selection*. MIT Press, Cambridge, MA, USA, 1992.

[MW88] M. Moore and J. Wilhelms. Collision detection and response for computer animation. *Computer Graphics*, 22(4), August 1988.

[PLM08] Riccardo Poli, William B. Langdon, and Nicholas F. McPhee. *A Field Guide to Genetic Programming*. http://www.lulu.com, 2008.

[Pre07] Paul Preece. Desktop tower defense 1.5 - a flash based puzzle / strategy game, 2007. `http://www.handdrawngames.com/DesktopTD/Game3.asp`.

[Sna10] Jan Snajder. The genprog package, 2010. http://hackage.haskell.org/package/genprog.