Small summery

Boris Prochnau

14. Juni 2014

Inhaltsverzeichnis

1 First meeting (14.04): Model of interest

These are some main points of the model we are using for the simulation. They are the product of the first meeting with Loren Coquille and Martina Baar.

- We have N traints and $(X_{i=1,\dots,N})$ are the amount of members of the traits.
- We use a constant mutation rate that does not depend on the traits μ.
- but the other rates are depending on traits and change with time (by as the amount of traitmembers rises)
  - b_i, [intrinsic birth-rate]
  - d_i, [intrinsic death-rate]
  - $\left(\sum_{j=1}^{N} \frac{c_{\cdot,j}}{K} x_j\right)_i$, [competition death]
  - $\mu\left(\frac{b_{i-1}+b_{i+1}}{2}\right)$, [extrinsic birth-rate]
- We use 3 groups of PPP's where every group represents an event.
  - $N_t^{b_i}$
  - $N_t^{d_i}$
  - $N_t^{\bar{c}_i}$

  where $\bar{c} = \left(\sum_{j=1}^{N} \frac{c_{\cdot,j}}{K} x_j\right)_i$. Therefore we get 3N processes running that compete about the first one occurring with each one triggering an death/birth for an trait.
- With respect to the fact that the distribution of the N_t changes with the size of the population we can think of an reset of the parameters of N_t due to the fact that the increments are exponentially distributed and therefore are memoryless (markov property)

- The use of coloring one trait was (only → ask later) to explain the superposition of the PPP associated with the simulation to extract the occured event from the PPP's

$$\rightarrow N_t^{total_i} = N_t^{b_i+d_i+\bar{c}_i} = N_t^{b_i} + N_t^{d_i} + N_t^{\bar{c}_i}$$

Therefore we would decide:

$$X_i^{total} = \begin{cases} \text{coloring d} & \text{with prob. p} \\ \text{coloring b} & \text{with prob. q} \\ \text{coloring } \bar{c} & \text{with prob. 1-p-q} \end{cases} \tag{1}$$

with:

$$p = \frac{d_i}{d_i + b_i + \bar{c}_i} \qquad q = \frac{b_i}{d_i + b_i + \bar{c}_i} \qquad 1 - p - q = \frac{\bar{c}_i}{d_i + b_i + \bar{c}_i}$$

2 Second meeting (22.04): Pseudocode

First thing to mention is that the Pseudocode snippets are marked as „Algorithm“, but in fact they are just functions and I don't know yet how to change the name in this particular LaTeX environment. Also I use a „=“ instead of „←“ because I think it provides better readability.
The following Pseudocode will contain many function-calls. Every function can be identified as a verb, and objects or local variables are nouns. There will also be one Boolean „isBorn“ which is an adjective.
Generally the functions have no return values because in this situation we tell functions to do something (not ask). Means we pass a task to them. There will be no requests for return values.
One other thing to mention is that the function names can get long, but they will (or should) always say what the function does. Not more or less.
The function-body should always follow a rule called „pretty much what you expected“. Some functions do violent this rule slightly. They are usually more than 6 lines and will be changed later.
First we start with the main Step of the Algorithm and continue with further explaining of the used functions. When a new function appear, it will be explained directly below it. Also we seperate this section in 3 parts, each one is dedicated to one of the functions used in this following EvolutionStep():

Algorithm 1 EvolutionStep()

Require: -
Ensure: A full evolution Step happened

```
1: calculateEventRates();
2: sampleEventTime();
3: changeATrait();
```

This function does a full evolution step. First thing to do is to calculate the Event rates with „calculateEventRates“, than we sample an exponential time with our current rate parameters with „sampleEventTime“ and finish the Step with changing a Trait with „changeATrait“.
Most functions-calls should not need to be explained, that's why I leave this explanations out in following comments.
This function will be improved later with not calculating the Rates in every new step, rather than updating them with the previous changes.

2.1 Calculating total-event rates

We will encounter 2 classes called „Trait“ and „Events“. They are so called Data Classes and usually mainly store data.
They are visible at all time and can be used like global variables.
In this first part of calculating total-event rates we only need to introduce the „Trait“ class which will be used as an array of traits „Trait[$i \leq n$]“ with n beeing the maximum number of traits. An Trait Object has attributes that can be accessed through a dot operator like „Trait[i].Members“ what would be „the Members of Trait i“.
This is a listing of all attributes in one Trait Object:
class Trait

- BirthRate
- TotalBirthRate
- DeathRate
- TotalDeathRate
- TotalTraitRate
- TotalEventRate - [static]
- CompDeathRate[i][j] - [static]

Here is much work with Superposition of the PPP included. We summarize the sum of birth-rates from members within a trait to the „TotalBirthRate“ (mutation included), same for „TotalDeathRate“ (competition included).
The „TotalTraitRate“ is the summed “TotalBirthRate“ and „TotalDeathRate“ and means the total rate of events for a specific trait.
„TotalEventRate“ and „CompDeathRate“ are static, this means that they are the same for all initialized Trait objects. They (all items from the array) access the same variable for this. „TotalEventRate“ is the same as usual and „CompDeathRate“ is the competition-matrix with according rates.

Algorithm 2 calculateEventRates()

Require: -
Ensure: All (total)Rates will be set
1: **for** i=0 **to** n-1 **do**
2: calculateTotalDeathRateOf(i)
3: **end for**
4: calculateTotalBirthRates(0);
5: calculateTotalEventRate();

In the next function we will manipulate attributes of the Trait Objects. The mutation will be accessed without and Traitindex: „Trait.Mutation“ because it is static and therefore the same for all Objects.

Algorithm 3 calculateTotalBirthRates(StartIndex: i)

Require: int i
Ensure: Total birthrate of Trait „i“ will be set (recursively)
1: Trait[i].TotalBirthRate = (Trait[i].Members)·(Trait[i].BirthRate)
2: **if** $i < n - 1$ **then**
3: calculateTotalBirthRates(i+1)
4: Trait[i].TotalBirthRate += $\frac{Trait.Mutation}{2} \cdot Trait[i+1].TotalBirthRate$
5: **end if**
6: **if** $i > 0$ **then**
7: Trait[i].TotalBirthRate += $\frac{Trait.Mutation}{2} \cdot Trait[i-1].TotalBirthRate$
8: **end if**

In Algorithm 3 in line 3 is used recursion, because this improves the calculation speed a lot, although it slightly makes code less intuitive.

Algorithm 4 calculateTotalDeathRateOf(TraitIndex: i)

[H]
Require: int i
Ensure: Total deathrate of Trait „i“ will be set
1: Trait[i].TotalDeathRate = 0;
2: addTotalIntrinsicDeathRateOf(i);
3: addTotalCompetitionDeathRateOf(i);

Algorithm 5 addTotalIntrinsicDeathRateOf(TraitIndex: i)

1: Trait[i].TotalDeathRate = (Trait[i].DeathRate) · (Trait[i].Members)

Algorithm 6 addTotalCompetitionDeathRateOf(TraitIndex: i)

```
1: for j=0 to n-1 do
2:     Trait[i].TotalDeathRate += (Trait.CompDeathRate[i,j])·(Trait[j].Members);
3: end for
```

Algorithm 7 calculateTotalEventRate()

Require: -
Ensure: Current Totaleventrate is set

```
1: for i=0 to n-1 do
2:     Trait[i].TotalTraitRate = Trait[i].TotalBirthRate
                                 + Trait[i].TotalDeathRate;
3:     Trait.TotalEventRate += Trait[i].TotalTraitRate;
4: end for
```

2.2 Sampling the next event-time

Here will appear a, not yet mentioned, object that will not be explained further, called Dice. The Dice Object will provide a uniform or exponential random Variable.

Algorithm 8 sampleEventTime()

Require: -
Ensure: First ringing Eventclock has been sampled

```
1: double Parameter = Trait.TotalEventRate;
2: double newEvent = this.Dice.RollExpDice(Parameter);
3: Events.EventTimes.push(newEvent);
```

Here we use Dice.RollExpDice(λ) to get $X \sim exp(\lambda)$. The same is possible for Dice.RollUnifDice(λ) to get $X \sim Unif[0, \lambda]$.

2.3 Changing a trait

Here we will work with the actual Events taking place. For this purpose i introduce the Events class like before the Trait class:
class Events

- Dice
- EventTimes[i]
- ChosenTrait[i]
- isBirth[]

The EventTime, ChosenTrait and isBirth are so called vectors. They are dynamic containers and we need to push an item into them to append it to the last entry.

Algorithm 9 changeATrait()

Require: -
Ensure: make a change to the Population with current Parameters

```
1: choseTraitToChange();
2: choseEventType();
3: executeEventTypeOnTrait();
```

Algorithm 10 choseTraitToChange()

Require: -
Ensure: Trait is chosen for changing

```
1: double Parameter = Trait.TotalEventRate;
2: double HittenTrait = Dice.rollUnif(Parameter);
3: for i = 1 to n-1 do
4:    if HittenTrait ≤ Trait[i].TotalEvent then
5:       this.ChosenTrait.push(i);
6:       break;
7:    end if
8:    HittenTrait -= Trait[i].TotalEvent;
9: end for
```

Algorithm 11 choseEventType()

Require: -
Ensure: Decision for Birth or Death is made

```
1: int i = Events.ChosenTrait.lastentry();
2: double EventType = Dice.rollUnif(Trait[i].TotalTraitRate);
3: if EventType ≤ Trait[i].TotalBirthRate then
4:    Events.isBirth.push(true);
5: else
6:    Events.isBirth.push(false);
7: end if
```

Algorithm 12 executeEventTypeOnTrait()

Require: -
Ensure: Chosen event will occur on chosen trait

```
1: if isBirth then
2:    Trait[ChosenTrait.lastentry()] += 1;
3: else
4:    if ChosenTrait.Members > 0 then
5:       Trait[ChosenTrait.lastentry()] -= 1;
6:    end if
7: end if
```

List of Algorithms

3 Thrid and Fourth meeting (Di 20.05 and 13.05): Graphical plans - afterwards

Zunächst habe nicht meine bisherigen Fortschritte präsentiert. Dazu gehörte ... Außerdem haben wir die graphischen Ansprüche an das Programm besprochen. Dabei wurden folgende Ergebnisse erzielt:

- Man sollte eine gute Möglichkeit haben die Parameter im Programm einsehen zu können und auch eine gute Möglichkeit Daten zu laden.
- Was besonders wertvoll ist, wäre eine Möglichkeit neue Instanzen zu generieren. Dabei hat sich jedoch das Problem gestellt dass es keine einfache Konsole mit simplem einseitigen Output und Input gibt, da man während einer Instanzgenerierung auch weiterhin mit dem Hauptfenster interagieren könnte. Das Problem wurde etwas umständlich über eine kontrollierte Deaktivierung der anderen Interfaceelemente gelöst.

Zur Darstellung des Graphen wurde folgendes gesagt:

- Es ist keine "Filmfunktion" gewünscht. Stattdessen sollte man einen Plot generieren und in diesen hereinzoomen und mit der Maus ziehen können.
- Außerdem ist eine Legende gewünscht die eine Übersicht über alle Graphen bietet.
- Um die Graphen später vergleichen zu können sollte es eine Möglichkeit geben den Graph zu speichern.

4 Fifth meeting (Fr 30.05): Improvements - afterwards

An diesem Tag habe bloß einige Änderungen vorgestellt und sichergestellt dass alle Funktionen ordnungsgemäß laufen:

4.1 Graphische Änderungen

Das waren hoffentlich alle optisch merklichen Änderungen:

- Ich habe eine Checkbox hinzugefügt die eine Option für sinnvollen Abbruch bietet. Diese Funktion prüft ob sich beide Populationen in einer /*epsilon* Umgebung um den erwarteten Zustand befindet und stoppt anschließend die Iterationen. Zu Gunsten der Übersicht werden noch mal 1/3 der Anzahl bisher gemachten Iterationen angefügt.
  Zu diesem Zeitpunkt ist $\epsilon = \frac{5}{\sqrt{K}}$.

- Es wurde ein Anzeigefehler behoben der für Geburts- und Todesraten immer nur die Populationsgrößen der Eigenschaften angezeigt hat.

- Auf dem Plotfenster habe ich ein Label mit den gemachten Iterationen hinzugefügt. Weiterhin findet sich dort jetzt auch ein Feld wo man einen Dateinamen eingeben kann der einem gespeichterten Bild dienen soll welches man mit dem neuen "save Image" Button erstellen kann. Das Bild wird zur Zeit als .png und .pdf gespeichert.

- Die x-Achsenbeschriftung lautet nun nicht mehr Millisekunden sondern "Time". Grund dafür ist dass ich nicht sicher bin welche Zeit unsere exponentielle Uhr ausgibt. Jedoch habe ich sie der Übersicht halber nicht mehr durch 1000 geteilt, da bei den gewünschten Instanzen ausreichend große Zeiten angenommen werden.

- Zuletzt wurde noch bei der Legende der für die k-te Erwartete Eigenschaft "k. Expected" zusätzlich der konvergierte Wert angezeigt "k. Exp: 1.048".

4.2 Interne Änderungen

- Ich habe mehrere Arten der Datenspeicherung ausprobiert. Zunächst habe ich die Daten alle nacheinander in eine "Storage" Datei geschrieben um schließlich einzeln die Daten mit "addData" zum Graphen hinzuzufügen. Das schien mir jedoch an der "addData" Stelle und am Datastream zu aufwendig. Danach bin ich auf eine alternative mit "Jumped Steps" umgestiegen. Die Jumped Steps basieren darauf das maximal 10.000.000 Punkte gespeichert werden und auch wenn viel mehr Iterationen gemacht werden, so werden nur Äquidistante 10 mio Stützstellen geplottet. Der Gedanke dahinter war die nicht Unterscheidbarkeit des Graphen für das menschliche Auge.

- Ich habe eine "GraphClass" hinzugefügt die alle Daten für das "PlotWindow" vorbereitet, so dass das Fenster nur noch darauf zugreifen muss. Damit habe ich eine Brücke zwischen den Fenstern und dem Plot geschlagen.
- Eine der wichtigsten Änderungen ist das Hinzufügen des QThreads. Dieser gewährleistet dass der Plot die Fenster nicht einfrieren lässt. Die Berechnung kann für mehr als 1mio Punkte bereits merkliche Zeit (¿3s) beanspruchen.
- ...to do...

4.3 bekannte Bugs

Es gibt hier noch einen Bug der mit der Beschriftung der x-Achse zu tun hat. Ich weiß noch nicht wie man ihn reproduziert, aber er könnte die Achse um einen großen Faktor zu groß skalieren. Der Plot weißt jedoch das geplante Verhalten auf und dieser Bug tritt nur bei ŕeplots; also dann auf wenn ein neuer Plot den alten überschreibt.

5 Sixth meeting (03.06): Introducing TSS

Einführung in TSS

- Das erste war das Einfügen einer Tabelle oder Grafik die einen guten Überblick über die Fitness der Population bietet. Diese gibt Aufschluss darüber ob TSS ordnungsgemäßes Verhalten zeigen wird. Allgemein ist es aber auch gut zu sehen welche Population wie stark auf eine andere wirkt, daher vielleicht schon ohne TSS sinnvoll.
  Um eine bessere Übersicht von ungewünschtem Verhalten zu zeigen werden Coexistenzen rot markiert. Später wird noch eine weitere grünliche Färbung angezeigt die eine Verdrängungswahrscheinlichkeit darstellt. Dieses Feature wird in beiden Programmen vorhanden sein.
  Die verwendete Formel zur Berechnung der Fitness war

  $$f_{x,y} = b_x - d_x - c_{x,y} \cdot \bar{n_x}$$

  mit:

  $$\bar{n_x} = \frac{b_x - d_x}{c_{x,x}}$$

  Demnach wird auf der Diagonale stets eine Null erwartet. Bei dieser Rechnung tritt jedoch eine Subtraktion auf die Auslöschung verursacht. Zu diesem Zweck habe ich die dargestellte Zahl auf 14 stellen runden lassen.

- Ein besseres Verständnis von TSS wäre nützlich. Dazu wurde mir aus [2] S.1135 empfohlen. Leider habe ich es bisher noch nicht geschafft es durchzuarbeiten. Es wird auf den nächsten Wochenplan kommen.

- Außerdem wurde die Skalierung der Mutationswahrscheinlichkeit mit K angepasst so dass TSS realistisch laufen kann. Dabei wurde der Faktor $\frac{1}{K^{1.5}}$ gewählt weil es den notwendigen Schranken die in [2] S.1135 beschrieben werden genügt. Ein weiterer Punkt den man optimieren kann wäre es eine Option zur Eingabe eines eigenen Exponenten für K wäre eine mögliche Erweiterung.

- In diesem Zustand ist es bereits Möglich einen TSS plot zu erzeugen. Jedoch wird die Zeit im Equilibrium tatsächlich berechnet was viel unnötige Rechenarbeit impliziert. Später solle eine lineare Interpolation bis zur ersten Mutation gemacht werden sobald das Equilibrium erreicht wird.

5.1 Bekannte Bugs

Es ist ein Bug bisher übrig. Bisher kenne ich noch nicht den Ursprung, aber er äußert sich (meistens) durch einen Programmabsturz nachdem ich eine kleinere Instanz lade und anschließend TSS1 bzw eine größere Instanz lade. Der Plot wurde dabei noch nicht ausgeführt.

6 Seventh meeting (10.06): Bug fixing and interpolation of Equilibrium

6.1 Was ist geplant?

Geplant ist:

- Beheben eines Fehlers beim Laden der Instanzen. Dieser wurde im letzten Kapitel beschrieben.
- Eine lineare Interpolation der Population im Equilibrium. Die Funktion dazu wurde bereits geschrieben.
- Korrekte Ausarbeitung eines geeigneten Abbruchkriteriums für TSS Prozesse.
- Punkte an denen eine Mutation stattgefunden hat sollen idealerweise hervorgehoben werden. Z.B. durch kreuze oder ähnliches.
- Am Besten wäre noch eine Skalierung der Zeitachse während des Equilibriums.
- Die Fitnessmatrix sollte bisher eine Bandmatrix sein weil es nur Mutationen zum Nachbarn gibt.

6.2 Was habe ich gemacht?

Konkret:

- Bisher habe ich eine Neudefinition der "isNear()" Funktion gemacht. Diese gibt es nun als "isNearDimorph()", "isNearMonomorph()" und "isNearTSS()". Diese tun was man vom Namen her erwartet, wobei "isNearTSS()" prüft die Nähe zum Equilibrium bzw. es prüft ob die Population bereits auf das Equilibrium getroffen ist (mit Rücksicht auf Auslöschung bis zu einer Genauigkeit von 10^{-10}) und ob die Population zur Zeit die einzig aktive ist.
- Weiterhin existiert eine neue Version der "iterateGraphPoint()" Methode. Sie heißt "iterateMutationPoint()" und soll den Zeitpunkt der nächsten Mutation der Nachbarn berechnen. Anschließend soll dieser Zeitpunkt in der Zeitlinie gespeichert werden und dort eine Mutation auf der Population ausgeführt werden.
- Diese "iterateMutationPoint()" Funktion soll anschließend von einer neuen Funktion "makeTSSIterations()" nur dann ausgeführt werden, wenn die neue "isNearTSS()" eine Bestätigung liefert. All diese Änderungen wurden in der GraphClass gemacht und werden in "generateEvolution()" verwendet.

- Ein wichtiger Zusatz für die Berechnung der Geburtenrate wurde getroffen. Dabei wurde jetzt nicht mehr zu beginn $b(i) \cdot N(i) + MutationFromNeighbors$ gerechnet, sondern $b(i) \cdot N(i) \cdot (1-\mu) + MutationFromNeighbors$. Das ist bisher nicht aufgefallen weil μ für gewöhnliche dimorphe Prozesse immer Null war.

- Eine sehr wichtige Änderung sollte die Berechnung des monomorphen stabilen Zustands gelten. Wenn eine positive Mutationswahrscheinlichkeit vorliegt, so sinkt die Arteigene Geburtenrate um die Mutationswahrscheinlichkeit. Somit berechnet sich das Equilibrium im TSS durch

$$\frac{b(x) \cdot (1-\mu) - d(x)}{c_{x,x}}$$

- NearTSS wurden die Grenzen geändert, so dass es jetzt eine Umgebung von 1/K um das Equilibrium gibt statt dem genauen Treffer. Das ist notwendig, weil es sein kann dass unser Equilibrium kein Vielfaches von 1/K ist, gewährleistet aber trotzdem die gröst mögliche Nähe zum Equilibrium.

- Es hat sich ein Bug ergeben der die Zeit falsch (bzw immer mit der ersten Rate) berechnet hat. Das wurde durch ein veraltetes "static" beim Ziehen der Würfelergebnisse verursacht.

- Ein weiteres Problem wurde behoben nach dem zufolge es noch keine ordentlich Anzeige für ein TSS gab, nachdem ein Trait zum ersten mal ausgestorben ist. Die Ursache lag in der "isNearTSS()" Funktion, daher wurde diese Überarbeitet.

- Es gab ein Problem weshalb der angezeigte stabile Zustand nicht bis zum Mutationspunkt durchgezogen wurde, sondern bereits sichtbar davor stoppte. Die Ursache war dass nur die Zeit bis zur nächsten Mutation als neuer Zeitpunkt gespeichert wurde, nicht aber die iterierte Zeit.

- Ich habe einen Bug behoben der das Programm manchmal zum Absturz brachte. Dabei handelte es sich wahrscheinlich um einen Speicherfehler, die genaue Ursache kenne ich nicht, aber ich konnte den Fehler beheben indem ich die Fitness Matrix "gecleart()" habe bevor sie erneut "resize()" aufruft.

todo

- Es sollten direkte getter und setter auf die Trait und Event Members ausgeführt werden satt sie temporär zu speichern wie in "generateEvolution()".

6.3 Bekannte Bugs

- Der letzte Punkt des Graphen wurde nicht gespeichert. Außerdem sollte der Graph auch wenn nichts mehr für ihn passiert, trotzdem bis zum ende

fortgesetzt werden. Dazu sollte der letzte Zeitpunkt nocheinmal mit dem Endzustand abgespeichert werden.

- Exp in der Legende sollte gesondert geändert werden.
- Problem mit K-member K-expected und Darstellung beider.

7 Zweites Bachelorseminar

Den größten Teil der Zeit seit dem letzten Seminar habe ich mit programmieren Verbracht. Daher wird ein größerer Teil am Ende dem Ergebnis dieser Arbeit gewidmet.

7.1 Ziel

Das Ziel meiner Bachelorarbeit ist es ein Programm zu entwickeln welches eine Simulation eines BPDL und TSS Prozesses durchführen kann. Dies soll dem besseren Verständnis über BPDL und TSS Prozessen dienen und kann zu statischen Nachweisen verwendet werden. Gerade deswegen ist es wichtig sicherstellen zu können dass sich keine Fehler in die Implementation einschleicht. Dazu jedoch mehr im Kapitel "Korrektheit der Implementation".

7.2 Model

Zunächst möchte ich das in der Simulation verwendete Model vorstellen.

- ? Dieses Modell soll eine Population beschreiben in der jedes Individuum ein Merkmal $x \in X \subset \mathbf{R}^n$ hat. Wobei die Simulation X ein endlichen diskreten Merkmalsraum verwendet der hier der Einfachheit halber als Indexmenge $X = \{1, \ldots, n\}$ repräsentativ für eine Durchzählung der Merkmale steht.
- Jedes Individuum kann sich entweder asexuell fortpflanzen oder sterben
- Diese Ereignisse treten dabei in exponentiell verteilter Häufigkeit auf. Diese exponentiellen Ereignisse haben die folgenden Raten:
  - b(x): Geburtenraten (Fortpflanzung) für ein Individuum mit Merkmal x.
  - d(x): natürliche Todesrate
  - c(x,y): Todesrate durch Wettbewerb zwischen zwei Individuen mit Merkmal x und y.
  - μ: Mutationswahrscheinlichkeit für eine Mutation "auf die Nachbarn" mit je 50% Wahrscheinlichkeit.

  Durch Ausnutzung des Superpositions Prinzips (? und der Markoveigenschaft) können mehrere exponentielle Ereignisse zusammengefasst werden, so dass man z.B. eine Todesrate und eine Arteigene Geburtenraten erstellen kann:
  - Arteigene Geburtenrate: $b(x) \cdot (1 - \mu) + (b(x + 1) + b(x - 1)) \cdot \frac{\mu}{2}$
  - Todesrate: $d(x) + \sum_{i=1}^{N_t} c(x, x_i)$, wobei N_t die Anzahl der Individuen zum Zeitpunkt t ist und x_i das Merkmal des i-ten Individuums. Diese Rate kann praktischer dargestellt werden als:

– Todesrate: $d(x)+\sum_{i=1}^{n} c(x,x_i)\cdot n_t(x_i)$, wobei n die Anzahl der Merkmale ist und $n_t(x_i)$ die Anzahl der Individuen mit Merkmal x_i zur Zeit t.

Die Simulation soll die Entwicklung der Merkmale und nicht die Ereignisse der Individuen darstellen, daher ist es unpraktisch diese zu betrachten. Alternativ werden die Ereignisse zu denen von Merkmalen zusammengefasst:

– Geburtenrate des Merkmals x:

$$B(x) = b(x)\cdot(1-\mu)\cdot n_t(x)+(b(x+1)\cdot n_t(x+1)+b(x-1)\cdot n_t(x-1))\cdot\frac{\mu}{2}$$

– Todesrate des Merkmals x:

$$D(x) = d(x)\cdot n_t(x) + n_t(x)\cdot\sum_{i=1}^{n} c(x,x_i)\cdot n_t(x_i)$$

- Um in der Simulation für den Fall vieler Merkmale nicht auch genauso viele Zufalls variablen ziehen zu müssen, werden die Raten zusammengefasst bis nur noch eine exponentielle Uhr läuft.

  – Ereignisrate des Merkmals x (Trait Rate):

  $$TR(x) = B(x) + D(x)$$

  – Totale Ereignis Rate (Total Event Rate):

  $$TER(x) = \sum_{x\in X} TR(x)$$

  Mit der Totalen Ereignisrate gibt es jetzt eine Rate die es erlaubt eine Zufallsvariable für das Eintreffen einer Variable zu ziehen. Anschließend ist es nur noch erforderlich mit der Ziehung zwei weiterer Zufallsvairablen festzustellen welchem Merkmal welches Ereignis zukommt.

- Die Population wird durch die ZV $\nu_t = \sum_{i=1}^{N_t} \delta_{x_i}$ beschrieben.
  Dabei nimmt ν_t Werte in $M_F(X) = \{\sum_{i=1}^{n} \delta_{x_i}\}$.
- ν_t ist ein Markov Sprung Prozess, $\int_x \mathbb{1}_y(x)\nu_t(dx)$.
- Der Generator des so definierten BPDL-Prozess ν_t ist:

$$L_{\phi(\nu)} = \int_x b(x)(1-\mu)[\phi(\nu+\delta_x)-\phi(\nu)]\nu(dx) \tag{2}$$

$$+ \int_x \int_{\mathbb{R}^d} b(x)(\mu)[\phi(\nu+\delta_{x+z})-\phi(\nu)]m(x,dz)\nu(dx) \tag{3}$$

$$+ \int_x d(x)[\phi(\nu-\delta_x)-\phi(\nu)]\nu(dx) \tag{4}$$

$$+ \int_x \left(\int_X c(x,y)\nu(dy)\right)[\phi(\nu-\delta_x)-\phi(\nu)]\nu(dx) \tag{5}$$

mit $\phi : M_F \to \mathbb{R}$

7.2.1 Normalisierung

7.2.2 Equilibrium

...

7.3 Algorithmus

Der verwendete Algorithmus führt einen Sprung des Markov Sprung Prozesses ab dem zuletzt bekannten Zeitpunkt t aus. Die tatsächlich verwendete Funktion sieht im Code so aus:

Algorithm 13 EvolutionStep()

Require: -
Ensure: A full evolution Step happened

```
1: calculateEventRates();
2: sampleEventTime();
3: changeATrait();
```

Etwas ausführlicher werden folgende Berechnungen angestoßen:

Algorithm 14 EvolutionStep()

Require: -
Ensure: A full evolution Step happened

```
1: —>calculateEventRates();
2: calculateTotalDeathRates()
3: calculateTotalBirthRates()
4: calculateTotalEventRate()
5: —>sampleEventTime();
6: sampleEventTime();
7: —>changeATrait();
8: choseTraitToChange();
9: choseEventType();
10: executeEventTypeOnTrait();
```

Schließlich der Ablauf der tatsächlichen Berechnung:

Algorithm 15 EvolutionStep()

Ensure: A full evolution Step happened
Require: $t, X = \{0, \dots, n-1\}$

—>calculateEventRates();
for $x \in X$ **do**
  $D(x) := n_t(x) \cdot \Big(d(x) + \sum_{y \in X} c(x,y) \cdot n_t(y)\Big)$
  $B(x) := \underbrace{b(x) \cdot (1-\mu) \cdot n_t(x)}_{arteigene}$
  if $x > 0$ **then**
    $B(x) += \underbrace{b(x-1) \cdot n_t(x-1)}_{MutationLinks} \cdot \frac{\mu}{2}$
  end if
  if $x < n-1$ **then**
    $B(x) += \underbrace{b(x+1) \cdot n_t(x+1)}_{MutationRechts} \cdot \frac{\mu}{2}$
  end if
  $TotalTraitRate(x) = B(x) + D(x)$
end for
$TotalEventRate := \sum_{x \in X} TotalTraitRate(x)$
—>sampleEventTime();
sample $Z \sim exp(TotalEventRate)$
$t += Z$
—>choseTraitToChange();
sample $Y \sim U(0, TotalEventRate)$
for $x \in X$ **do**
  if $Y \leq TotalTraitRate(x)$ **then**
    $ChosenTrait := x$
    break
  end if
  $Y -= TotalTraitRate(x)$
end for
—>choseEventType();
sample $Y \sim U(0, TotalTraitRate(ChosenTrait))$
if $Y \leq B(ChosenTrait)$ **then**
  isBirht := true
else
  isBirth := false
end if
—>executeEventTypeOnTrait();
if isBirth **then**
  $n_t(ChosenTrait) += 1$
else
  if $n_t(ChosenTrait) \geq 0$ **then**
    $n_t(ChosenTrait) -= 1$
  end if
end if

7.4 Simulation

Nun kommen wir zur eigentlichen Simulation. Damit ist sowohl die Darstellung als auch die Programmarchitektur gemeint.

7.4.1 Aufgabenteilung und Flexibilität

Die Architektur des Programms kann grob in drei Module gefasst werden. Die Idee der getrennten Aufgabenbereiche geht darauf zurück dass eine möglichst große Unabhängigkeit zwischen Arbeitsschritten notwendig ist um das Programm flexibel zu halten und sogenannten "Coderot" - "faulen Code" zu verhindern. Das bedeutet dass das Programm mit steigender Komplexität zunehmend unflexibler wird, also das Hinzufügen weiterer Features oder das Ändern/Verbessern zu sogenanntem "undefiniertem Verhalten" führt. (kurze Erklärung zum Begriff).

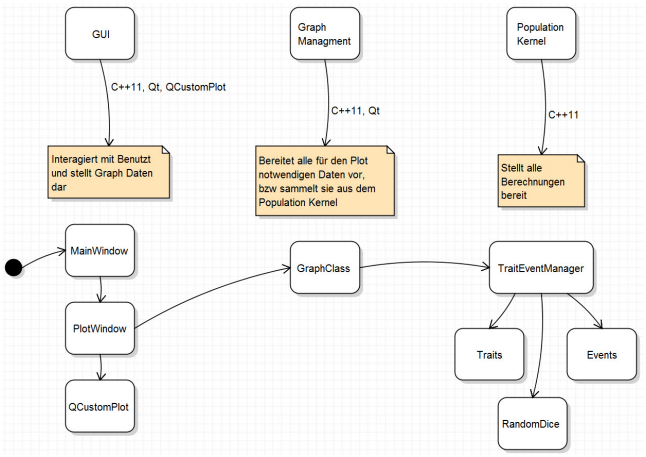

Abbildung 1: Arbeitsmodule und Klassenabhängigkeiten

Innerhalb der Arbeitsschritte sollte dabei genau die selbe Regel der Unabhängigkeit gelten wie bei den Modulen.

7.4.2 Layout

- Die Bedienung des Programms sollte das lesen und Anzeigen der Merkmals-Parameter bereitstellen. Da es viele Parameter gibt und die Anzahl der Parameter quadratisch mit der Anzahl der betrachteten Merkmale steigt, bietet sich das Lesen aus zuvor beschriebenen Dateien an.

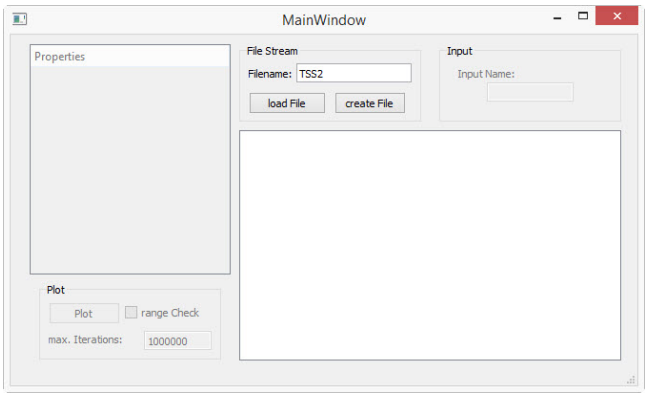

Abbildung 2: MainWindow nach dem Start

Zur Darstellung der gelesenen Parameter habe ich ein Baumstruktur gewählt.

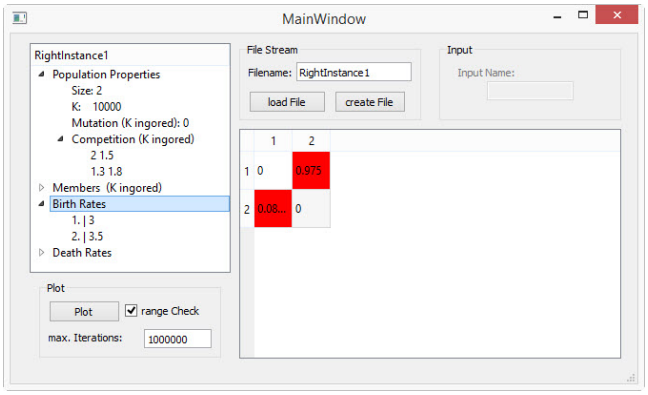

Abbildung 3: MainWindow mit geladenen Parametern

- Die letzte Herausforderung bestand darin eine Instanz durch das Programm geleitet erstellen zu können. Während diese Aufgabe bei einer Konsolenanwendung (bekannt aus den klassischen c Programmen) denkbar einfach mit "printf" und "scanf" erledigt werden konnte, sollte bei einem GUI eine Lösung her die Inputkonflikte verhindert und das Einlesen der Daten denkbar einfach macht. Dafür war es sinnvoll "Enter" als Bestätigung abzufangen und sicherzustellen dass der Curser nur innerhalb des gewünschten Feldes bleibt.

Was die Darstellung der Graphen angeht, hat sich ein einfaches Bild des fertigen Plots durchgesetzt, wobei das Zoomen und ziehen des Bildes notwendige Elemente zur Untersuchung des Graphen sind. Zusätzlich ist es notwendig viele Bilder vergleichen zu können. Hierfür wurde das Abspeichern des Bildes

gewünscht.
Die Simulation wird gestartet nachdem man auf den "Plot" Button drückt. Dabei wird ein neues Fenster geöffnet.

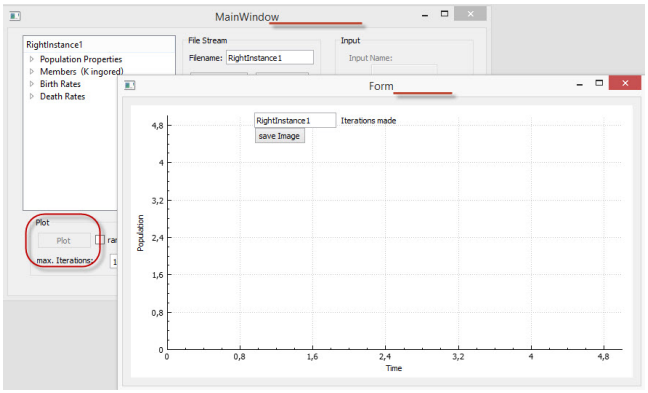

Abbildung 4: Start des PlotWindow

Sobald der Plot Button gedrückt wurde und das Fenster angezeigt wird, arbeitet die Simulation bereits im Hintergrund in einem eigenen Thread. Dort werden alle notwendigen Iterationen bzw. Sprünge der Population durchgeführt ohne den Hauptthread damit zu belasten. Während dieser Arbeiterthread aktiv arbeitet wird der "Plot" Button ausgegraut um mehrfaches Auslösen zu vermeiden und um anzuzeigen dass die Rechnung im Gange ist.
Der Arbeiterthread gewährleistet nicht nur eine flüssige Interaktion mit dem Programm, sondern verhindert auch effektiv dass das Betriebssystem denkt das Programm wäre Abgestürzt oder würde nicht mehr ordnungsgemäß funktionieren. Das würde sonst folgendes evtl. bekannte Bild hervorrufen:

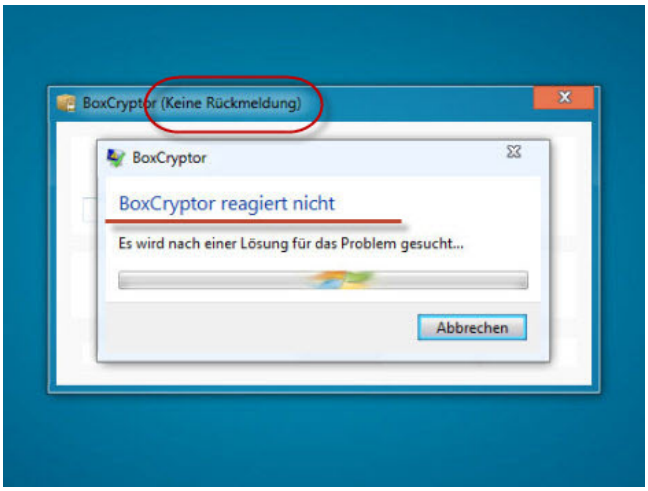

Abbildung 5: Hauptthread wurde überlastet

Wenn die Simulation einen gewünschten Zustand erreicht hat, oder die maximale Anzahl an gewünschten Iterationen absolviert hat, werden anschließend maximal 10mio Punkte auf dem Koordinatensystem zu Graphen verbunden. Das kann so aussehen:

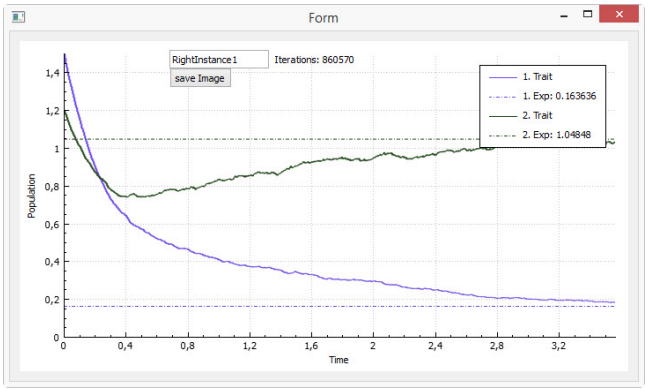

Abbildung 6: PlotWindow mit Dimorpher Population

7.5 Korrektheit der Implementation

(Korrektheit des Algorithmus nicht notwendig bzw möglich) Ein ganz besonders interessantes Thema ist die Korrektheit der Implementation. Diese ist generell mit steigender Komplexität schwerer zu prüfen (besonders bei Zufallsbedingten Simulationen).
Daher habe ich das Prinzip der "Testgetriebene Entwicklung" (Test Driven De-

veleopment) verwendet.

```
51  void TraitEventManagerTest::verifyWrittenData()
52  {
53      QCOMPARE(TraitClass::Size,3.);
54      QCOMPARE(TraitClass::Mutation,0.1);
55      for(int i = 0; i < TraitClass::Size; ++i){
56          QCOMPARE(Manager.Trait[i].BirthRate,10.);
57          QCOMPARE(Manager.Trait[i].DeathRate,5.);
58          QCOMPARE(TraitClass::CompDeathRate[i][i],2.);
59      }
60  }
61
62  // -------------------------- section 1: Rates -----------
63  /// Unit Tests for INPUT VALIDATION
64
65  void TraitEventManagerTest::readAndClearStandardInput()
66  {
67      Manager.initWithFile("ValidateTests.txt");
68      verifyWrittenData();
69      Manager.clearData();
70      QVERIFY(TraitClass::Size == 0.);
71      QVERIFY(Manager.Trait.size() == 0.);
72      QVERIFY(TraitClass::CompDeathRate.size() == 0.);
73      QVERIFY(Manager.Trait.size() == 0.);
74  }
```

Abbildung 7: UnitTest versichert korrektes lesen von Parametern aus Datei

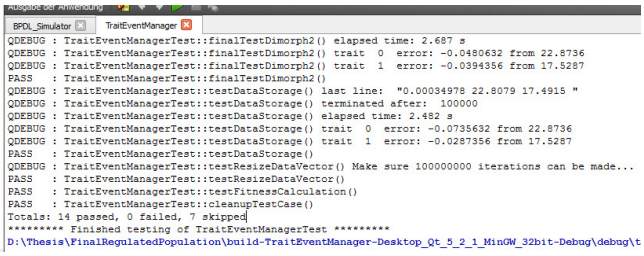

Abbildung 8: Ergebnise einiger Tests

Spätere statische Möglichkeiten

7.6 TSS

7.7 Warum ist es vielleicht schwierig simple Mechanik einzupflegen?

Wie man sich vielleicht denken kann sind viele kleine Features an und für sich sehr simpel. Unglücklicherweise gibt es viele kleine solche Features und Daten die mit dem Programm verbunden wären.

8 Achtes Treffen

8.1 Was ist geplant?

8.2 Was habe ich gemacht?

8.3 Bekannte Bugs

9 Weitere geplante Punkte

9.1 große Verbesserungen

- Es wäre sinnvoll wenn es eine Möglichkeit gibt die angestrebten stabilen Zustände als Option auszuwählen welche angezeigt werden sollten. Dabei sollte die Beschriftung "Dimorph" oder "Monomporph" darin auftauchen.
- dynamisches Abbruchkriterium welches für alle TSS Instanzen angemessen terminiert.
- Dokumentation zu Programmcode und Bedienung
- Anpassen der neuen Geburten und Equilibrien.
- Evtl. Abfrage beim Überschreiben von Dateien
- Instanzexplorer
- GraphClass aufräumen (RangeCheck etc.)
- Unit Tests anpassen und erweitern
- Beim speichern die Zeit hinzufügen um das häufige überschreiben zu vermeiden.
- Fehler beim Laden nicht definierter Instanzen prüfen!
- Unterschiedlich große Sprünge erklären (Auslöschung bei Addition -Nulladdition- oder Jumps?)
- Release Version des Programms erstellen.
- Es wurde gewünscht dass irgendwo die Anzahl der Mutationen bis ein Switch von dominanten Merkmalen stattgefunden hat angezeigt wird.

9.2 kleine Verbesserungen

- Items in der Tabelle sollten zentrieren
- Es sollte das aktuell verwendete K in der Simulation angezeigt werden.
- stabiler Zustand weniger Aufdringlich (z.B. größerer Abstand zwischen Strichen und geringere Strichstärke)

- Feedback beim Speichern eines Bildes
- Wenn Plot Button grau ist, soll
- Überschrift bzw Namen der Fenster ändern.