Small summery

Boris Prochnau

23. April 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 $\rightarrow$ 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 dont 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

```
calculateEventRates();
sampleEventTime();
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, thats 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

for i=0 **to** n-1 **do**
    calculateTotalDeathRateOf(i)
end for
calculateTotalBirthRates(0);
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

Trait[i].TotalDeathRate = 0;
addTotalIntrinsicDeathRateOf(i);
addTotCompetitionDeathRateOf(i);

Algorithm 5 addTotalIntrinsicDeathRateOf(TraitIndex: i)

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

Algorithm 6 addTotalCompetitionDeathRateOf(TraitIndex: i)

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

Algorithm 7 calculateTotalEventRate()

Require: -
Ensure: Current Totaleventrate is set

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

```
double Parameter = Trait.TotalEventRate;
double newEvent = this.Dice.RollExpDice(Parameter);
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

```
choseTraitToChange();
choseEventType();
executeEventTypeOnTrait();
```

Algorithm 10 choseTraitToChange()

Require: -
Ensure: Trait is chosen for changing

```
double Parameter = Trait.TotalEventRate;
double HittenTrait = Dice.rollUnif(Parameter);
for i = 1 to n-1 do
    if HittenTrait ≤ Trait[i].TotalEvent then
        this.ChosenTrait.push(i);
        break;
    end if
    HittenTrait -= Trait[i].TotalEvent;
end for
```

Algorithm 11 choseEventType()

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

```
int i = Events.ChosenTrait.lastentry();
double EventType = Dice.rollUnif(Trait[i].TotalTraitRate);
if EventType ≤ Trait[i].TotalBirthRate then
    Events.isBirth.push(true);
else
    Events.isBirth.push(false);
end if
```

Algorithm 12 executeEventTypeOnTrait()

Require: -

Ensure: Chosen event will occur on chosen trait

```
if isBirth then
    Trait[ChosenTrait.lastentry()] += 1;
else
    if ChosenTrait.Members > 0 then
        Trait[ChosenTrait.lastentry()] -= 1;
    end if
end if
```

List of Algorithms