Agentenbasiertes Programmieren in Netlogo

Georg Jäger, Universität Graz

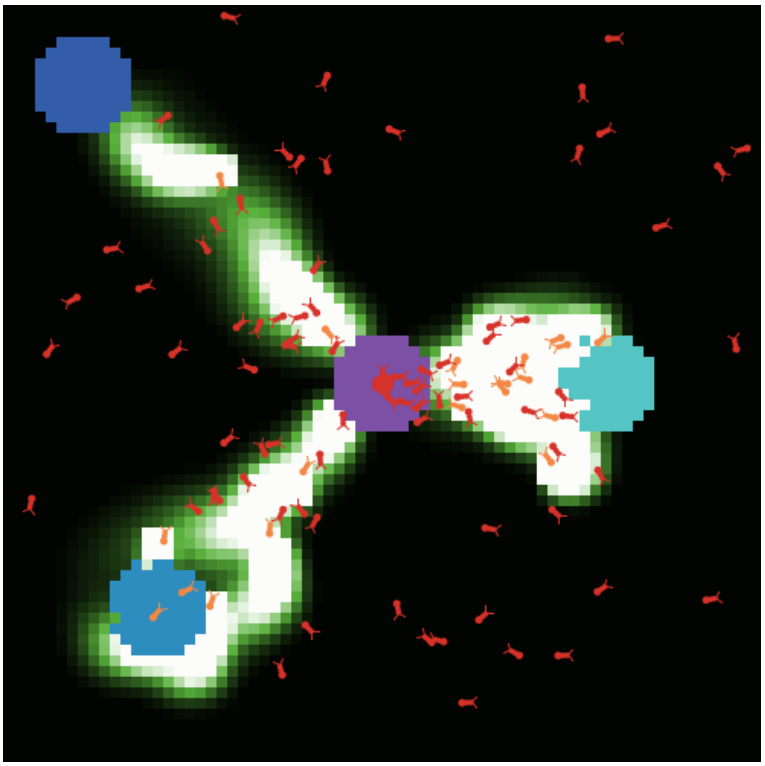
www.jaeger-ge.org

Inhaltsverzeichnis

Was ist agenten-basiertes Programmieren?

Agenten-basiertes Modellieren ist ein Ansatz zum Modellieren von komplexen Systemen, der ausnutzt, das viele System viel einfacher **bottom-up** beschrieben werden können. So ist das Verhalten einer Ameisenkolonie beispielsweise sehr schwer in Formeln und Gleichungen zu fassen, die Regeln, denen einzelne Ameisen folgen, sind aber sehr einfach. Mit Computerunterstützung ist es somit einfach möglich aus solchen Regeln in der **Mikro-Ebene** auf überaschende Resultate in der **Makro-Ebene** zu kommen.

Um einen Einblick in Netlogo zu bekommen ist es am einfachsten ein bestehendes Modell zu öffnen. Dazu klickt man auf File → Models library und öffnet das Modell Biology → Ants. Der Bildschirm hat sich nun mit einigen Objekten gefüllt: Links oben sind Knöpfe und Schieberegler, darunter befindet sich ein Plot und rechts eine schwarze Fläche. Wenn man nun auf den Knopf Setup drückt wird das Modell initialisiert. Die Welt, die bislang nur schwarz war füllt sich mit Ameisen (roter Punkt in der Mitte) und Futterquellen (Farbige Kreise). Die Simulation beginnt, wenn man den Knopf Go betätigt. Die Ameisen laufen in eine zufällige Richtung auf der Suche nach Futter. Ameisen die zufällig ein Futterstück gefunden haben tragen es zurück und geben dabei Pheromone ab, die den anderen Ameisen helfen, die Futterquelle zu finden. Der Plot stellt grafisch dar wie viel Nahrung noch in welcher Futterquelle ist und zeigt somit den Erfolg der Ameisenkolonie an. Mit den Schiebereglern kann man gewisse Parameter der Simulation verändern, also etwa die Ameisenpopulation.



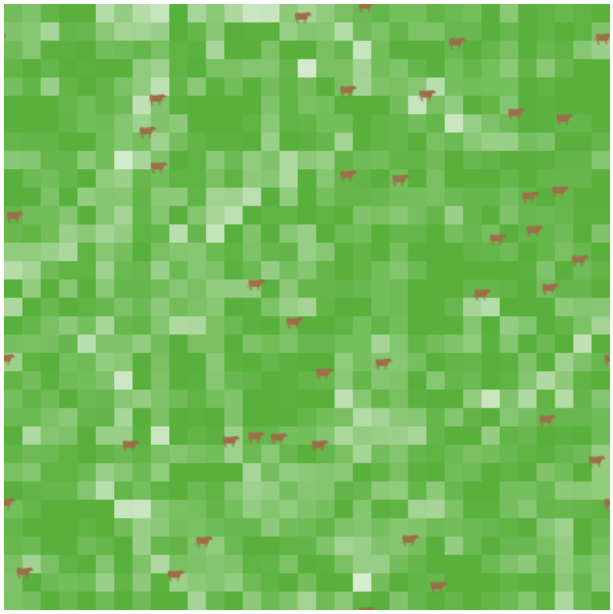
Sind wir nur am Ausführen von Modellen interessiert, ist diese Ansicht schon ausreichend. Ohne Programmierkenntnisse ist es somit möglich fertige Modelle zu verwenden. Wir wollen aber lernen eigene Programme zu erstellen, bzw. existierende anzupassen. Das eigentlich Programmieren passiert in Netlogo in einer anderen Ansicht: Wechselt man vom Reiter Interface in den Reiter Code sieht man den Programmiercode, der hinter der Ameisensimulation steckt. Vereinfacht gesagt wird hier erklärt was welche Knopf und welcher Schieberegler bewirkt, wie sich zum Beispiel die Ameisen bewegen und so weiter. Mit dem genauen Code und der Programmiersprache werden wir uns aber erst später beschäftigen, fürs erste ist es wichtig die Grundprinzipien von Netlogo zu verstehen.

Mit Netlogo erstellt man agentenbasierte Modelle. Im Beispielprogramm sind die Agenten die Ameisen, also eigenständige Wesen, die für sich Entscheidungen treffen, ein spezielles Verhalten haben und miteinander und mit ihrer Umwelt interagieren. Programmiertechnisch sind Agenten Objekte, die gewisse Eigenschaften haben. Alle Agenten habe eine Position, eine Farbe, eine Form, eine Richtung und eine Größe. Zusätzlich können Agenten je nach Modell weitere Eigenschaften haben. Diese Agenten leben immer in einer Welt.

Die Welt in Netlogo ist meist ein Quadrat. Die Größe kann von Modell zu Modell unterschiedlich sein, standardmäßig ist der Punkt ganz links unten der Punkt (-16,-16) und der Punkt ganz rechts oben der Punkt (16,16). Die Welt ist also 33 x 33 Felder groß. Diese Felder werden Patches genannt. Patches haben, ähnlich wie Agenten, gewisse Eigenschaften, z.B. eine Farbe. Ja nach Modell können Patches auch weitere Eigenschaften haben. Im Ameisenmodell gibt es zum Beispiel Patches mit Futter, und Patches ohne Futter, und die Patches haben zum Beispiel auch eine gewisse Pheromonkonzentration. Jeder Agent befindet sich immer auf einem Patch, ähnlich wie eine Schachfigur auf einem Schachbrett. Es gibt jedoch einen wichtigen Unterschied: Bei Schachfiguren ist es egal, wo auf dem Feld sie stehen. Agenten haben aber immer eine exakte Position, das heißt sie befinden sich nicht immer in der Mitte eines Patches, sondern können an einem beliebigen Punkt sein. Somit können sich auch mehrere Agenten auf dem gleichen Patch befinden. Um eigene Modelle zu erstellen, oder bestehende Modelle anzupassen müssen wir die Programmiersprache lernen, die Netlogo verwendet. Die Befehle und Kommandos heißen meist anders, aber die groben Strukturen (wie Schleifen und If-Abfragen) sind sehr ähnlich wie in anderen Programmiersprachen. Am besten funktioniert der Einstieg in diese Programmiersprache, indem wir ein ganz einfaches Modell von Grund auf erstellen.

1 Modell 1 - Die Kuhweide

In diesem Modell lernen wir die Grundlagen von Netlogo kennen. Wir werden Agenten erstellen, die dann mit ihrer Umwelt interagieren und sie so beeinflussen, was wiederum Auswirkungen auf die Agenten selbst hat. Als möglichst einfaches Beispiel verwenden wir Kühe (Agenten) die auf einer Kuhweide (Umwelt) leben.



Im ersten Schritt erstellen wir einen Setup-Knopf. Dieser wird am Beginn der Simulation gedrückt und initialisiert Agenten und Umwelt. Im Code-Fenster fügen wir die Befehle hinzu, die ausgeführt werden sollen, wenn der Knopf gedrückt wird. Um diese Zuordnung zwischen Code und Knopf zu machen, schreiben wir `to setup` und `end`. Zwischen diese beiden Ausdrücke kommen die Befehle, die ausgeführt werden sollen, wenn jemand Setup drückt. Wir beginnen mit `clear-all`, was alle Agenten, Variablen und sonstige Reste von vorherigen Modelldurchläufen löscht. Als nächstes möchten wir die Weide erstellen. Sie wird durch grüne Patches symbolisiert. Immer wenn wir in Netlogo Befehle an Patches oder Agenten senden wollen, müssen wir sie mit `ask` ansprechen. Um beispielsweise alle Patches anzusprechen schreiben wir `ask patches [...]`. In die eckigen Klammern kommt dann der Befehl, der an jeden einzelnen Patch gehen soll. In unserem Fall möchten wir die Patchfarbe des Patches auf grün setzen. Eigenschaften, wie beispielsweise die Patchfarbe (in Netlogo pcolor genannt) ändert man immer mit dem Befehl `set` gefolgt von der Eigenschaft, die man ändern möchte. Für uns also `set pcolor green`. Abschließend schreiben wir noch `reset-ticks` um die Netlogo-Zeit auf 0 zu setzen. Die gesamte Setup-Prozedur sieht nun so aus:

```
to setup
   clear-all
   ask patches[
      set pcolor green
   ]
   reset-ticks
end
```

Als nächstes werden wir Agenten in die Simulation einbauen. Dazu definieren wir zuerst einen sogenannten breed, also eine Art oder Gattung von Agenten. Das ist vor allem wichtig, wenn man mehrere Arten von Agenten hat, zum Beispiel Wölfe und Schafe, die ein völlig anderes Verhalten haben sollen. Breeds definiert man am Anfang des Codes mit `breed [name-in-mehrzahl name-in-einzahl]` also für uns:

```
breed [cows cow]
```

Sobald der Breed cow einmal definiert ist, können wir neue Agenten mit dem Befehl `create-cows anzahl []` erstellen. Wir beginnen mit einer einzigen Kuh, die wir im Setup erstellen:

```
create-cows 1 []
```

Wenn wir das Programm nun testen, sehen wir dass die Kuh als Standardagent, also als Pfei mit einer zufälligen Farbe in der Mitte der Netlogowelt erstellt wird. Möchten wir unseren Agenten individualisieren, können wir in den [ ]-Block nach `create-cows` Befehle schreiben, die af den neu erstellten Agenten angewandt werden. Mit dem `set` Befehl können wir auch Farbe (color) und Form (shape) verändern.

```
create-cows 1 [
   set color brown
   set shape cow
]
```

Um den Agenten an einer zufälligen Position starten zu lassen, können wir auch x- und y-Koordinate (xcor und ycor) des Agenten festsetzen. Zufällige x- und y-Koordinaten erstellt man mit `random-xcor` bzw. `random-ycor`:

```
create-cows 1 [
   set color brown
   set shape cow
   set xcor random-xcor
   set ycor random-ycor
]
```

Nun möchten wir das Modell ausbauen und mehrere Kühle gleichzeitig erstellen. Dazu könnten wir einfach die Zahl nach make-cows im Code ändern. Eleganter wird es aber, wenn wir ein Interfaceelement einbauen, in dem man die Startpopulation sozusagen von außen einstellen kann, ohne im Code etwas ändern zu müssen. Wir erstellen dazu einen Slider, also einen Schieberegler. Wir geben ihm den Namen `population`. Netlogo kennt dieses Wort nun als Variable, setzt also jedes mal, wenn im Code `population` steht, die Zahl ein, die bei dem Schieberegler gerade eingestellt ist. Wir ändern unseren Code auf

```
create-cows population [
...
```

um eine variable Anzahl an Kühen erstellen zu können.

Nun ist der Setup-Prozess fertig. Als nächstes kümmern wir uns um die Zeitentwicklung. Wir erstellen einen Knopf mit Namen Go und aktivieren die Option forever. Dadurch wird die Prozedur go nicht nur ein einziges Mal durchgeführt (wie das Setup) sondern unendlich oft hintereinander. In die go Prozedur schreiben wir, was in jedem Zeitschritt passieren soll. Das Grundgerüst dazu sieht so aus:

```
to go
   tick
end
```

Der Befehl tick zählt die Netlogo-Zeit um einen Zeitschritt nach vorne. Sonst passiert in unserem Modell noch nichts. Um Bewegung einzubauen müssen wir einen Bewegungsbefehl an alle Kühe schicken. Gleich wie bei den Patches, machen wir das mit `ask`. Der Befehl, um einen Patch vorwäts zu gehen lautet `forward 1`.

```
to go
   ask cows [
      forward 1
   ]
   tick
end
```

Wenn wir das Modell testen sehen wir, dass die Kühe immer geradeaus gehen. Wenn Sie am Rand der Netlogowelt anstoßen, erscheinen sie wieder auf der andern Seite. Um die Bewegung ein wenig realistischer zu gestalten können wir die Agenten zufällig nach rechts und nach links drehen. Die Befehle dazu heißen `right` und `left`. Die Grad, um die gedreht werden soll können wir zufällig machen: Der Befehl `random 21` erzeugt eine zufällige ganze Zahl von 0 bis inklusive 20 (also 21 verschiedene Zahlen).

```
to go
   ask cows [
      right random 21
      left random 21
      forward 1
   ]
   tick
end
```

Als nächstes möchten wir einbauen, dass die Kühe einen Teil von dem Gras auf dem aktuellen Feld fressen. Dadurch wird der Patch ein bisschen heller, bis er fast weiß ist. Hierzu benötigen wir eine neue Struktur: Die If-Abfrage. Die If-Abfrage ist eine Wenn-Dann-Bedingung, die einen Befehl nur dann ausführt, wenn eine gewisse Bedingung gegeben ist. In unserem Fall: Ein patch soll nur dann seine Farbe ändern, wenn eine Kuh dort steht. Als Befehl `any? cows-here`. Um einen Farbton in Netlogo zu ändern, kann man einfach eine Zahl addieren um ihn heller zu machen, oder eine Zahl abziehen um ihn dunkler zu machen. `green + 0.5` ist also ein bisschen heller als `green`.

```
ask patches[
   if any? cows-here[
      set pcolor pcolor + 0.5
   ]
]
```

Um zu verhindern, dass die Patches weißer als weiß werden, sollten wir noch eine If-Abfrage einbauen, die überprüft ob die Farbe eines Patches über einem von uns definierten Grenzwert liegt. Wenn dem so ist, wird die Farbe des Patches wieder auf den Grenzwert zurückgesetzt. Wir entscheiden uns für die Farbe 59, ein sehr helles Grün:

```
ask patches[
   if any? cows-here[
      set pcolor pcolor + 0.5
      if pcolor >= 59 [
         set pcolor 59
      ]
   ]
]
```

Kühe, die auf einem Feld stehen, dass nun schon ganz hell ist finden keine weitere Nahrung und sterben. Der Befehl, der Agenten aus der Simulation entfernt heißt `die`. Agenten haben neben der Eigenschaft color (die Farbe die sie selbst haben) auch die Eigenschaft pcolor (die Farbe des patches auf dem sie stehen). Festzustellen ob die Agenten also auf einem hellgrünen Feld (Farbe 59) stehen ist also nicht schwer.

```
ask cows [
   if pcolor = 59 [
      die
   ]
]
```

Wenn wir die Simulation nun laufen lassen, sehen wir, dass die Kühe nach einiger Zeit aussterben, weil nicht mehr genügend Nahrung vorhanden ist. Um diese Entwicklung besser verfolgen zu können, bauen wir einen Plot ein, also ein Diagramm, dass uns den zeitlichen Verlauf einer Größe (hier die Anzahl der Kühe) darstellt. Wir ändern den Text im Plotfenster auf `plot count cows` um die Anzahl der Kühe anzeigen zu lassen. So können wir die zeitliche Veränderung gut beobachten.

Abschließen möchten wir nun auch, dass das Gras langsam wieder nachwächst. In jedem Zeitschritt wird das Gras ein klein bisschen dunkler, und zwar um 0.02. Außerdem brauchen wir eine If-Abfrage, die verhindert, dass die Farbe zu dunkel wird. Immer wenn die Farbe unter 55 sinken würde, soll sie wieder auf 55 gesetzt werden. Alle Befehle, die man zur Umsetzung dieser Aufgabe benötigt kennen wir bereits.

AUFGABE 1.1

Alle Patches sollen angesprochen werden.
Sie sollen ihre Patchfarbe um 0.02 reduzieren.
Wenn die Patchfarbe danach unter 55 ist, soll sie auf 55 gesetzt werden.

Nun können wir schon interessantes Verhalten beobachten. Wenn zu viele Kühe auf der Weide sind, dauert es nicht lange, bis die ersten sterben. Es ist aber auch möglich, dass sich ein Gleichgewicht einstellt, und die überlebenden Kühe immer genügend Nahrung finden. In jedem Fall gibt der Plot eine genaue Auskunft über die zeitliche Entwicklung unserer Simulation.

Zusammenfassung

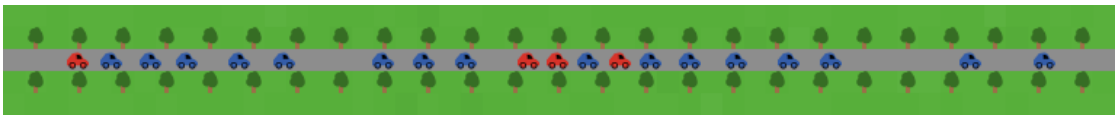
- Eine Setupprozedur beinhaltet immer die Befehle `clear-all` und `reset-ticks`.
- Breeds von Agenten erstellen wir mit `breed name-in-mehrzahl name-in-einzahl`.
- Neue Agenten vom Breed cow erstellt man mit `make-cows anzahl[]`
- Mit `set` kann man Attribute von Agenten und Patches (wie color, shape, xcor) ändern: `set name-des-attributs neuer-wert-des-attributs`
- If-Abfragen ermöglichen, dass Befehle nur unter gewissen Umständen ausgeführt werden
- Der Befehl `any? cows-here` meldet wahr, wenn sich auf dem aktuellen Patch mindestens ein Agent vom Breed cow befindet, sonst meldet er falsch.

Expert Knowledge

Genaugenommen passiert bei einem `ask` mehr, als man hier sieht. Zuerst erstellt Netlogo eine Liste von allen Objekten, die angesprochen werden (also z.B. alle Patches). Dann werden alle Objekte in dieser Liste in einer zufälligen (!) Reihenfolge angesprochen. Die Objekte handeln also nach der Reihe und nicht gleichzeitig. Das heißt die Aktionen des ersten Agenten könnten alle weiteren Agenten sogar noch im gleichen Zeitschritt beeinflussen. Durch die Zufallsauswahl gibt es aber keinen ausgezeichneten Agenten, und all diese Effekte der gegenseitigen Beeinflussung teilen sich gleichmäßig auf alle Agenten auf.

Netlogo arbeitet mit sogenanntem Kontext um zu unterscheiden welche Befehle gerade erlaubt sind. Der Kontext unterscheidet sozusagen wer gerade am Zug ist: Einer der Patches, einer der Agenten oder niemand, in Netlogosprache der Observer. Im Observerkontext kann man zum Beispiel neue Agenten mit `make` erstellen, dieser Befehl ist im Agenten-Kontext und im Patch-Kontext verboten. Umgekehrt benutzten wir im Beispiel einmal den Befehl `cows-here` während gerade ein Patch durch `ask patches` angesprochen wird. Der Befehl `cows-here` wäre im Observer-Kontext, also wenn gerade niemand am Zug ist, verboten. Logisch, denn wenn niemand angesprochen wird ist auch nicht klar, wo here überhaupt sein soll. Gerade bei umfangreichen Programmen ist es in Netlogo sehr wichtig, immer zu wissen in welchem Kontext man sich gerade befindet, wer also gerade (von wem) angesprochen wird. Es kann nämlich vorkommen, dass man einen Patch anspricht, der einen Agenten ansprechen soll, der wiederum einen weiteren Agenten anspricht.

2 Modell 2 - Phantomstau



In diesem Modell möchten wir Fahrzeuge auf ganz einfache Weise simulieren und die Frage der Stauentstehung beantworten.

Wir beginnen wie immer mit einem setup Knopf:

```
to setup
   clear-all
   reset-ticks
end
```

Als nächstes bauen wir eine ganz einfache Straße auf. Dazu benutzen wir eine Prozedur, die wir setup-road nennen. Innerhalb der setup-Prozedur schreiben wir einfach setup-road, was wir damit meinen definieren wir direkt darunter:

```
to setup
   clear-all
   setup-road
   reset-ticks
end

to setup-road
   ask patches [
      if pycor = 0 [ set pcolor white ]
]
```

Wir malen also einfach eine weiße Linie, die eine Fahrspur darstellen soll. Außerdem ändern wir die Größe der Netlogowelt, indem wir auf die Welt rechtsklicken und unter “edit” den Wert für die maximale y-Koordinate auf 4, und den der maximalen x-Koordinate auf 25 stellen. Zusätzlich zur Straße brauchen wir aber auch noch Fahrzeuge. Wir definieren also den breed car. Zusätzlich zu den vorgefertigten Netlogo-Eigenschaften (shape, color,...) brauchen unsere Agenten aber auch noch eine Variable, die die momentane Geschwindigkeit speichert. Solche Variablen, werden Agenten-Variablen genannt und werden wie folgt definiert

```
breed [cars car]
cars-own [ speed ]
```

Zunächst erstellen wir nur ein Auto, das von links nach rechts fährt :

```
to setup
   clear-all
   setup-road
   setup-cars
   reset-ticks
end
to setup-cars
   create-cars 1 [
      set shape "car"
      set color blue
      set xcor random-xcor
      set ycor 0
      set heading 90
      set speed 0.1 + random-float 0.2
   ]
end
```

Das Auto hat die Farbe blau und startet auf einer zufälligen x-Koordinate. Die y Koordinate ist fix vorgegeben. Heading 90 bedeutet, es schaut nach rechts. Am Schluss setzen wir die Geschwindigkeit auf einen zufälligen Wert zwischen 0.1 und 0.3, indem wir den Befehl random-float 0.2 benutzen, der uns eine zufällige Fließkommazahl zwischen 0 und 0.2 liefert.

Wenn wir die Simulation jetzt laufen lassen, tut sich noch nicht viel. Speed ist in diesem Fall einfach nur eine Variable, die aber noch nichts bewirkt. Legen wir einen Go-Button an und schreiben wir die go-Prozedur, damit die Autos auch wirklich fahren.

AUFGABE 2.1

Alle cars sollen angesprochen werden.
Sie sollen sich vorwärts bewegen (forward).
Die Schrittlänge soll `speed` betragen.

Nun fährt das Auto im Kreis, immer mit der gleichen Geschwindigkeit. Spannender wäre es aber, wenn sie bis zu ihrer Maximalgeschwindigkeit beschleunigen würden. Das können wir ganz einfach einbauen:

```
to go
   ask cars [
      set speed speed + 0.001
      if speed > 0.9 [ set speed 0.9 ]
   fd speed
   ]
tick
end
```

Die Fahrzeuge beschleunigen also immer um einen gewissen Betrag, wenn sie ihre Maximalgeschwindigkeit erreichen beschleunigen sie nicht weiter. Im nächsten Schritt wollen wir weitere Fahrzeuge einbauen. Dazu ist es aber notwendig Kollisionen zu vermeiden. In der Theorie ist das ganz einfach: Solange man langsamer fährt als das Fahrzeug vor einem, kann es zu keiner Kollision kommen. Wir müssen unsere Beschleunigung-Prozedur also ein wenig anpassen: Wenn vor uns ein Fahrzeug ist fahren wir ein bisschen langsamer als das Fahrzeug vor uns:

```
to go
   ask cars [
      set speed speed + 0.001
      if any? cars-on patch-ahead 1[
         set speed [speed] of one-of cars-on patch-ahead 1
         set speed speed - 0.02
      ]
      if speed > 0.9 [ set speed 0.9 ]
      if speed < 0 [ set speed 0]
   fd speed
   ]
tick
end
```

Nach dem Beschleunigen wird überprüft ob ein Agent auf dem Feld direkt voraus ist (genau genommen auf dem Feld das 1 Feld in Fahrtrichtung entfernt ist): `any? cars-on patch-ahead 1`. Wenn dem so ist, muss die Geschwindigkeit angepasst werden auf die des vorausfahrenden Fahrzeuges. Die Geschwindigkeit eines Fahrzeuges auf diesem spezifischen Patch bekommen wir mit `[speed] of one-of turtles-on patch-ahead 1` Im Anschluss reduziert das Fahrzeug seine Geschwindigkeit noch um einen kleinen Betrag. Außerdem darf die Geschwindigkeit nicht größer sein als die Maximalgeschwindigkeit und nicht kleiner als 0.

Auf diese Weise können wir Kollisionen verhindern. Dem Einbauen von mehreren Fahrzeugen steht nun also nichts mehr im Wege. Wir erstellen einen Slider mit den Namen carnumber die von 1 bis 40 gehen können und adaptieren die setup-cars-Prozedur:

```
to setup-cars
   create-cars carnumber [
      set shape "car"
      set color blue
      set xcor random-xcor
      set ycor 0
      set heading 90
      set speed 0.1 + random-float 0.2
   ]
end
```

Um besser zu sehen, wann Fahrzeuge anhalten müssen, können wir die Fahrzeuge, die vor sich jemanden erkennen rot einfärben:

```
...
set color blue
set speed speed + 0.001
if any? cars-on patch-ahead 1[
   set color red
   set speed [speed] of one-of cars-on patch-ahead 1
   set speed speed - 0.02
]
...
```

Wir können die Simulation nun testen. Was uns interessiert ist die mittlere Geschwindigkeit der Autos, andererseits auch der zeitliche Durchschnitt dieser Geschwindigkeit. Dazu erstellen wir einen plot der die mittlere Geschwindigkeit der Fahrzeuge wie folgt plottet:

```
plot mean [speed] of cars
```

Nun können wir verschiedene Fahrzeugzahlen einstellen und analysieren, wie sich die Geschwindigkeit verhält. Mit ein bisschen Herumprobieren wird man feststellen, dass die Fahrzeuge bis zu einer gewissen Dichte mit ihrer Maximalgeschwindigkeit fahren können. Bei mehr Fahrzeugen kommt es aber zu einem Stau. Interessant wäre es jetzt genau diese kritische Anzahl der Fahrzeuge zu finden. Dazu müsste man die Simulation mit vielen verschiedenen Startbedingungen laufen lassen, das Ergebnis aufschreiben und dann daraus eine Grafik machen. Glücklicherweise gibt es für genau so ein Problem ein eigenes Tool, das in Netlogo eingebaut ist. Es nennt sich BehaviorSpace und macht genau das: Es führt die Simulation mit ganz vielen Startbedingungen automatisch aus und speichert die Resultate. Wir starten BehaviorSpace mit Tools → BehaviorSpace und klicken unten auf "new" um ein neues Experiment zu starten. Standardmäßig sind alle Parameter genau so eingestellt wie sie momentan im Modell sind. Wir lassen alles gleich, nur die carnumber Zeile ändern wir auf:

```
["carnumber"[1 1 40]]
```

Das heißt wir wollen als Fahrzeugzahl nicht einen fixen Wert haben, sondern alle Werte zwischen 1 und 40 mit einer Genauigkeit von 1. (also 1,2,3,...,39,40) Im zweiten großen Textfeld können wir definieren, welche Größe wir als Endergebnis einer Simulation

speichern wollen. In unserem Fall würde sich die Größe

```
mean [speed] of cars
```

anbieten, also die mittlere Geschwindigkeit der Fahrzeuge. Ganz unten können wir noch ein Zeitlimit festlegen, nachdem jeder Simulationslauf abgebrochen wird. 10000 Schritte sollten ausreichen. Außerdem müssen wir noch das Häkchen von "measure runs at every step"**entfernen**, damit pro Durchlauf nur ein Messpunkt (am Ende der Simulation) gespeichert wird. Wir bestätigen mit OK und starten das Experiment mit Run. Netlogo bietet uns table oder spreadsheet output an, wir wählen table und bestätigen mit OK. Schon fängt BehaviorSpace mit dem Simulieren an und speichert alle Ergebnisse in ein csv-file, also eine Tabelle. Die Ergebnisse, die in dieser Tabelle gespeichert sind können dann mit einem beliebigen Programm visualisiert werden. Wer gerne Excel verwendet kann die Daten über "Daten" → "Aus Datei" importieren. Dabei ist es wichtig das Komma als Trennzeichen zu deklarieren. Außerdem ist der Dezimaltrenner von Excel standardmäßig ein Komma und kein Punkt, wie in der Wissenschaft üblich. Und auch Zahlen in wissenschaftlicher Darstellung (z.b. 1.234E-7) werden oft nicht oder nicht richtig erkannt. Alternativen zum Erstellen von Grafiken sind gnuplot, oder natürlich das matplotlib package von Python.

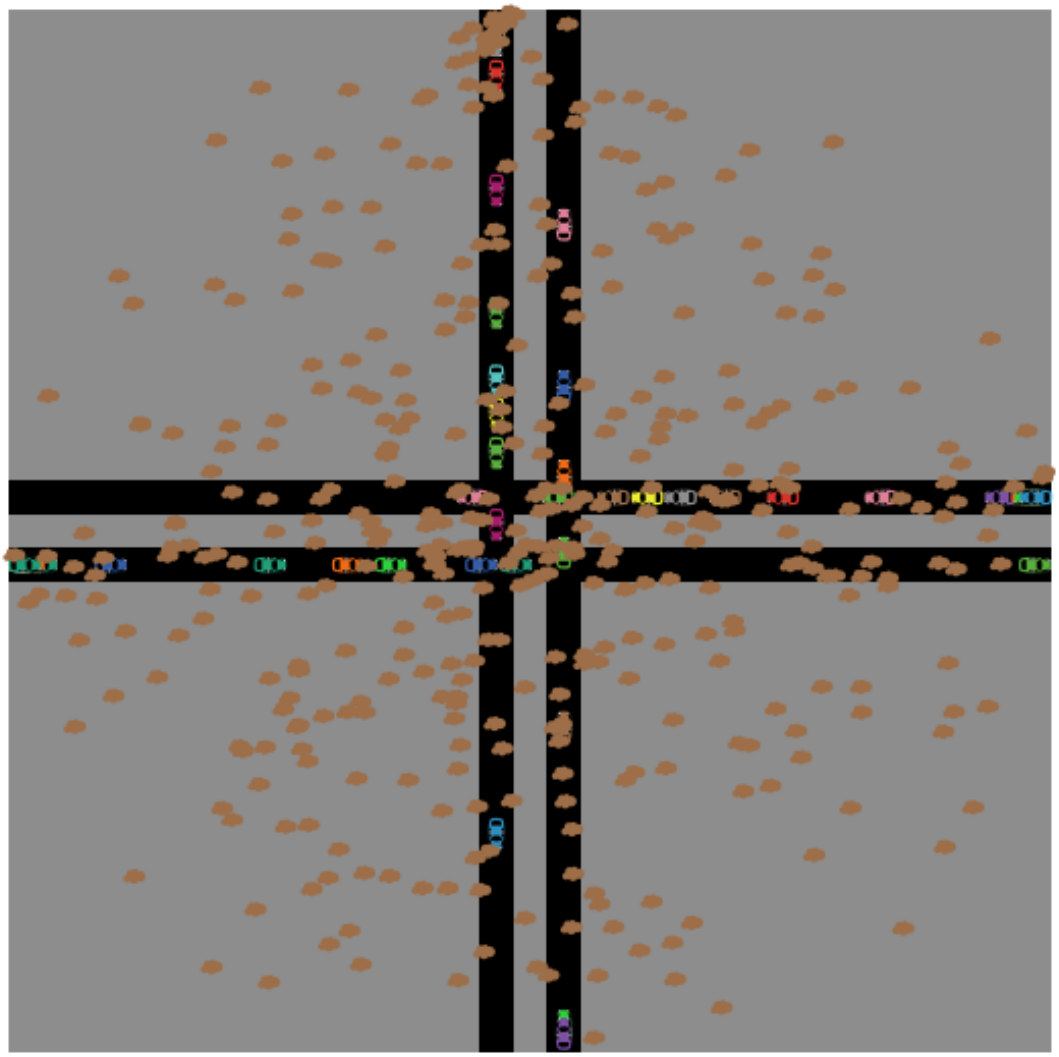
Zusammenfassung

- Eigene Agenteneigenschaften (z.B. die Eigenschaft speed für cars) können mit `cars-own [speed]` definiert werden
- Um festzustellen ob sich ein Agent vom breed car auf einem Patch befindet kann man `any? cars-on` verwenden
- Der patch direkt vor dem aktuellen Agenten wird mit `patch-ahead 1` ausgewählt
- Um Agenteneigenschaften von einem Agenten, der momentan nicht am Zug ist zu bekommen, schreiben wir die Eigenschaft in eckige Klammern gefolgt von `of` und der Information um welchen Agenten es geht
- Um einen zufälligen Agenten aus einer Menge auszuwählen kann man den Befehl `one-of` verwenden.

Expert Knowledge

Behavior Space ist ein leistungsstarkes Tool mit dem man einfach eine große Anzahl an Prozessoren gleichzeitig verwenden kann, ohne dass das eigentliche Programm parallelisierbar sein muss. Gerade für Sensitivitätsanalysen oder systematisches Auswerten von Parametern ist Behavior Space deswegen sehr nützlich.

3 Modell 3 - Feinstaub



In diesem Modell betrachten wir die räumliche und zeitliche Verteilung von Feinstaub in einer Modellstadt. Die Stadt wird sehr vereinfacht sein, trotzdem wird man aber schon gewissen Effekte erkennen können.

Wir beginnen damit, das Fundament der Stadt einzuzeichnen: Die Stadt selbst wird grau, sie bekommt zwei schwarze Straßen und das Einfärben vom Rand der Stadt wird später einige Dinge vereinfachen. Um Patches mit einer gewissen Eigenschaft auszuwählen gibt es den Befehl `with`. Außerdem möchten wir in diesem Modell nicht, dass die Welt periodische Randbedingungen hat, was wir in den Weltoptionen (world wraps) einstellen können.

```
to setup
   ca
   reset-ticks
   ask patches [
      set pcolor gray]
   ask patches with [pxcor = -1 or pxcor = 1][
      set pcolor black
   ]
   ask patches with [pycor = -1 or pycor = 1][
      set pcolor black
   ]
   ask patches with [pxcor = 16 or pxcor = -16 or pycor = 16 or pycor = -16]
      [set pcolor white]
end
```

Um die Straßen zu füllen, hätten wir gerne Autos. Zuerst erstellen wir ein einziges Auto, und lassen es sich bewegen, später werden wir beide Prozesse ausbauen.

```
breed [cars car]
to setup
...
   create-cars 1 [
      setxy 1 -15
      set shape "car top"
      set heading 0]
end
to go
   movecars
   tick
end
to movecars
   ask cars[
      fd 0.1]
end
```

Wenn das Fahrzeug das Ende der Straße erreicht (weißer Bereich) soll es aus der Simulation entfernt werden.

```
to removecars
   ask cars with [pcolor = white][
      die]
end
```

Das Erstellen der Fahrzeuge lagern wir nun vom Setupprozess in einen eigenen makecars Prozess aus. Hier besteht eine gewisse Chance in jedem Tick, dass ein neues Fahrzeug aus einer der 4 Himmelsrichtungen kommt. Eine Chance von z.B. 1% (also 10 Tausendstel) können wir einbauen, indem wir eine Zufallszahl mit `random 1000` erzeugen und dann fragen, ob die Zahl klener ist als 10. Das trifft nämlich genau auf 10 der

1000 möglichen Zahlen zu, da alle gleich wahrscheinlich sind haben wir eine Chance von 10 Tausendstel. Für die zufällige Richtung legen wir eine lokale Variable an. Das geht mit `let namedervaribalen wertdervariablen`. Wir erzeugen eine Zahl mit `random 4` und benutzen dann If-Abfragen um das Auto auf der richtigen Seite der Netlogowelt entstehen zu lassen.

```
 to makecars
   if random 1000 < 10[
      let direction random 4
      if direction = 0 [
         create-cars 1 [
            setxy -1 15
            set shape "car top"
            set heading 180]
      ]
      if direction = 1 [
         create-cars 1 [
            setxy 1 -15
            set shape "car top"
            set heading 0]
      ]
      if direction = 2 [
         create-cars 1 [
            setxy 15 1
            set shape "car top"
            set heading -90]
      ]
      if direction = 3 [
         create-cars 1 [
            setxy -15 -1
            set shape "car top"
            set heading 90]
      ]
   ]
end
```

Um den Fahrzeugen ein realistischeres Verhalten zu geben, geben wir ihnen eine Geschwindigkeit und benutzen ähnliche Regeln wie im Staumodell.

AUFGABE 3.1

Alle cars sollen angesprochen werden. Sie sollen ihren speed um 0.001 erhöhen.
Wenn vor ihnen ein anderes Fahrzeug ist (`any? other car-on patch-ahead 1`) sollen sie ihren speed auf 0 setzen.
Danach sollen sie sich mit ihrem momentanen speed vorwärts bewegen.

Wir wollen nun einen ganzen Tag simulieren, und das Verkehrsaufkommen relativ realistisch darstellen. Die Simulation soll um Mitternacht starten und bis Mitternacht des nächsten Tages laufen. Im Interface soll man die Fahrzeugmenge einstellen können. Die meisten Fahrzeuge soll es zwischen 7 und 9 und zwischen 16 und 18 Uhr geben. Ein tick soll einer Sekunde entsprechen. Wir erstellen einen Schieberegler caramount und verbessern die Prozedur makecars. Dabei ist es hilfreich, dass wir If-Bedingungen über `and` verknüpfen können.

```
to makecars
   let carchance caramount / 10
   if ticks >= 60 * 60 * 7 and ticks <= 60 * 60 * 9[
      set carchance caramount
   ]
   if ticks >= 60 * 60 * 16 and ticks <= 60 * 60 * 18[
      set carchance caramount
   ]
   if ticks >= 60 * 60 * 11 and ticks <= 60 * 60 * 13[
      set carchance caramount / 4
   ]
   if random 1000 < carchance[
      let direction random 4
(...)
```

Bei viel Verkehr kann es hier zu einem kompletten Stau kommen, da die Kreuzung nicht geregelt ist. Um das zu verhindern, reicht es einer Fahrtrichtung Vorfahrt zu geben. Alle Autos, die nach Norden fahren, dürfen sich immer bewegen. Das bauen wir so ein:

```
to movecars
   ask cars[
   set speed speed + 0.001
   if any? other cars-on patch-ahead 1 and heading != 0[
      set speed 0
   ]
   forward speed
]
```

Jetzt kommen wir zum Kern der Simulation: Die Fahrzeuge erzeugen Feinstaub, wenn sie fahren. Wir beginnen ganz einfach: in jedem Tick, hat jedes Fahrzeug eine Chance von 0.1% ein Feinstaubpartikel zu erzeugen. Feinstaub behandeln wir als Agent mit eigener breed. Sie bewegen sich zufällig und werden entfernt, wenn sie die Stadtgrenze erreichen. Dazu brauchen wir einen neuen Befehl: Wir wollen, dass ein Agent einen weiteren Agenten erzeugt. Dafür eignet sich der Befehl `hatch`, der eine exakte Kopie des Agenten erstellt, der gerade am Zug ist. Anschließend kann man in eckigen Klammern Änderungen durchführen, um aus einem geklonten Auto einen Feinstaubpartikel zu machen.

```
to makefinedust
   ask cars[
   if random 1000 < 1 [
      hatch 1[
         set breed finedusts
         set color brown
         set shape "cloud"
         set size 0.7
   ]]]
   ask finedusts[
      right random 360
      fd 0.1
      if pcolor = white[
         die]
   ]
end
```

Nun können wir den zeitlichen Verlauf der Feinstaubbelastung in einem Plot mitverfolgen, indem wir einfach die Anzahl der finedusts plotten. Man sieht gleich: durch die Zufallsbewegung des Feinstaubs, bleibt der meiste Feinstaub bis Mitternacht in der Stadt, obwohl kein neuer produziert wird. Glücklicherweise gibt es in der Realität (meist) Wind, der einen großen Einfluss auf die Feinstaubverteilung hat. Erstellen wir einen Schieberegler von 0 bis 0.01 für die Windgeschwindigkeit, der in unserer Stadt von Ost nach West bläst. Dadurch werden alle finedusts ein wenig verschoben:

```
set xcor xcor + wind
```

Wir sehen, der Wind hat einen sehr großen Einfluss auf die Bewegung von Feinstaub (und auch anderen Emissionen) und sollte nicht vernachlässigt werden.

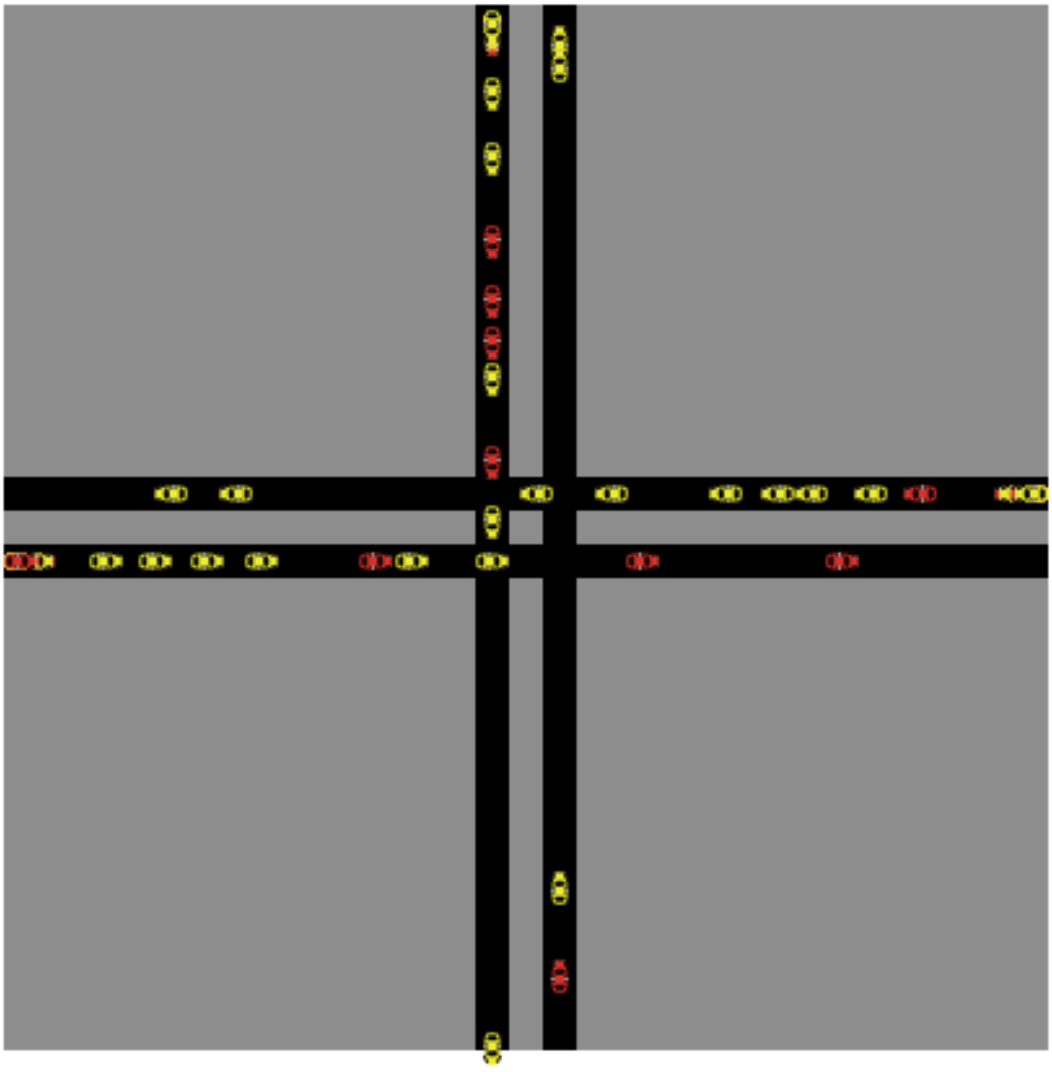
Zusammenfassung

- Um Patches oder Agenten mit einer gewissen Eigenschaft auszuwählen verwendet man den Befehl `with`
- Solche Abfrage, so wie auch If-Abfragen kann man mit `and` und `or` verknüpfen
- Um zufällige Chancen einzubauen erstellt man am besten eine Zufallszahl und überprüft, ob sie kleiner ist als die gewünschte Chance. Eine 10% Chance sieht also so aus: `if random 100 < 10`
- Lokale Variablen kann man mit `let name wert` anlegen
- Wenn ein Agent einen weiteren Agent erstellen soll verwendet man den Befehl `hatch`, der eine exakte Kopie erstellt, die dann angepasst werden kann.

Expert Knowledge

Der Grund, warum wir in diesem Modell einen Hatch-Befehl brauchen und nicht einfach mit `create-finedusts` arbeiten können, ist, dass die Befehle von Netlogo immer nur in einem gewissen Kontext funktionieren. Der Befehl create-finedusts funktioniert nur im Observer-Kontext. Sobald ein Agent am Zug ist, funktioniert der Befehl nicht und wir müssen auf `hatch` ausweichen. Wenn kein Agent, sondern ein Patch einen Agenten erstellen soll heißt der Befehl dazu `sprout`.

4 Modell 4 - CO2



In dieser Übung verändern wir das Feinstaubmodell, um ein CO2-Modell zu erhalten. Dann bauen wir Elektroautos ein und berechnen die effektiven Emissionen bei unterschiedlichem Energiemix.

AUFGABE 4.1

Vom Feinstaubmodell sollen folgende Dinge entfernt werden:
Der Wind-Schieberegler
Der breed fnedusts
Die Prozedur make-finedust
Der Plot finedust

AUFGABE 4.2

Eine globale Variable namens co2 soll angelegt werden, die später die gesamten CO2-Emissionen mitprotokollieren soll.
Globale Variablen legt man mit `globals [name]` am Anfang des Programms an.

AUFGABE 4.3

Die Variable co2 soll jedes Mal erhöht werden, wenn sich ein Auto vorwärts bewegt hat (also direkt nach der Zeile mit `forward`).
Der Wert der Abgase soll errechnet werden aus den üblichen Abgaswerten (120 g/km +/- 10 g/km) mal 0.01, da ein Patch in unserem Modell 10 Meter lang ist, also 1% (0.01) von einem Kilometer.
Normalverteilte Zufallszahlen bekommt man mit `random-normal mittelwert standardabweichung`.

AUFGABE 4.4

Nach einem Tag (= 60 * 60 * 24 ticks) soll die Simulation automatisch beendet werden. Dazu kann man eine If-Abfrage direkt innerhalb der go-Prozedur verwenden, die die momentane Tickzahl (`ticks`) überprüft und die Simulation dann mit `stop` beendet.
Es soll einen Plot geben, der den momentanen Wert von co2 zeigt.

AUFGABE 4.5

Es soll einen Schieberegler namens `e-chance` geben, der zwischen 0 und 100 einstellbar ist.
Beim Erstellen von Fahrzeugen soll es eine e-chance-% chance geben, dass das neue Fahrzeug gelb wird (Elektroauto), sonst wird es rot.
Das Addieren von co2 soll nur mehr für rote Fahrzeuge passieren.

AUFGABE 4.6

Es soll eine globale Variable namens e-emissions geben, die ausdrückt, dass durch die Produktion von elektrischem Strom auch CO2 emittiert wird.
Es soll einen chooser geben, indem man den Energiemix von Graz, Wien oder den USA wählen kann. Ja nach Wahl wird dann im setup ein anderer Wert für e-emissions gesetzt. Die genauen Werte dafür werden im unteren Absatz erklärt. Gelbe Autos sollen nun auch die Variable co2 erhöhen, und zwar um das, was unter e-emissions gespeichert ist multipliziert mit ihrem momentanen speed.

Der Strom in Graz kommt vereinfacht gesagt aus 90% Wasserkraft 5% Wind und 5% Biomasse. Aus einem LCA lässt sich errechnen, wie viel CO2 emittiert werden, um 1 MWh Strom zu produzieren:
Wasserkraft: 10 kg
Windkraft: 30 kg
Biomasse: 50 kg
Wenn wir nun davon ausgehen, dass ein Elektroauto auf einem Kilometer etwa 0.2 kWh verbraucht, können wir auch die indirekten Emissionen eines Eautos überschlagen. Dabei müssen wir nur ein wenig mit den Einheiten aufpassen: wir möchten alles in Gramm und kWh und haben:
Wasserkraft: 10 kg /MWh = 10 g/ kWh
Also kommen wir auf

```
set e-emissions 0.2 * (0.9 * 10 + 0.05 * 30 + 0.05 * 50)
```

Das sind unter 3 g pro Kilometer, also sehr viel weniger als Verbrennungsmotoren, aber dennoch nicht 0.
Anders ist, die Situation leider mit einem anderen Strommix. Zum Beispiel der Mix aus Wien:
45% Wasser (10g/kWh), 45% Erdgas (500g/kWh), 5% Wind (30g/kWh), 5% Biomasse (50g/kWh)
Das führt zu:

```
0.2 * (0.45 * 10 + 0.45 * 500 + 0.05 * 30 + 0.05 * 50)
```

also fast 50 g/km, das ist schon fast die Hälfte eines Verbrennungsmotors. Und dabei sind Effekte wie Ladeverluste (bis zu 20%) und die energieintensive Produktion von Elektroautos noch nicht eingerechnet.
Sehen wir uns abschließen noch die USA an. Hier kommt der Strom aus:
37% Öl (700g/kWh), 30% Gas (500g/kWh), 15% Kohle (800g/kWh), 10% Nuklear (20g/kWh), 8% „renewable“ (30g/kWh)
Das führt zu:

```
0.2 * (0.37 * 700 + 0.30 * 500 + 0.15 * 800 + 0.10 * 20 + 0.08 * 30)
```

Hier landen wir bei über 100 g pro km.

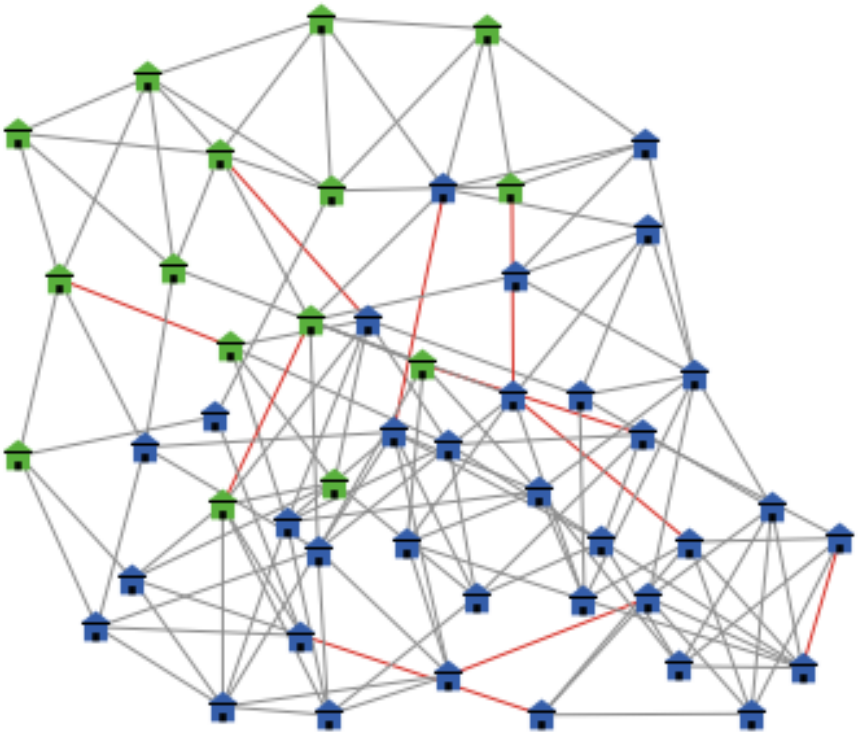
Zusammenfassung

- Globale Variablen können wir mit `globals [name]` anlegen.
- Normalverteilte Zufallszahlen erzeugt man mit `random-normal mittelwert standardabweichung`.
- Die aktuell vergangene Anzahl der ticks bekommt man mit `ticks`
- Prozeduren (z.B. die go-Prozedur) können mit `stop` beendet werden.

Expert Knowledge

Auf globale Variablen kann jeder Agent, jeder Patch und jede Prozedur immer zugreifen und den Wert auch verändern. Globale Variablen sind in Netlogo wichtiger als in anderen Programmiersprachen, da die starke Kontextabhängigkeit oft eine globale Variable nötigt macht, wo man in anderen Programmiersprachen einfach den Wert der Variablen an die Funktionen übergeben könnte. Glücklicherweise macht das verwenden von globalen Variablen Netlogo nicht wirklich langsamer.

5 Modell 5 - Technologiediffusion



In diesem Modell betrachten wir wie eine neue Technologie diffundiert, das heißt wie sie sich in einem System ausbreitet. Als Beispiel wählen wir ein kleine Stadt und betrachten Heiztechnologien. Wir erstellen ein neues Netlogomodell mit einem setup und einem go Knopf. Die go Prozedur bleibt fürs erste leer:

```
to go
end
```

In der setup Prozedur wird die Stadt erstellt. Dieser Prozess ist aber relativ aufwendig, deswegen lagern wir ihn in eine eigene Prozedur mit dem Namen build-city aus. Des Weiteren färben wir die Patches weiß.

```
to setup
   clear-all
   ask patches[set pcolor white]
   build-city
   reset-ticks
end
```

Die build-city Prozedur generiert die Agenten, in unserem Fall die Häuser. Wir erstellen also einen neuen breed mit `breed [houses house]` In unserem Modell möchten wir die Häuser als Netzwerk darstellen, die Häuser sollen also verbunden sein. Um in Netlogo mit Netzwerken arbeiten zu können, benötigen wir die Erweiterung nw, die sich mit folgendem Befehl einbinden lässt:

```
extensions [nw]
```

Danach können wir in unserem Programm neue Befehle benutzen, die es ohne diese Erweiterung nicht gäbe. Ein solcher Befehl, der das Erstellen von Netzwerken sehr einfach macht, heißt `nw:generate-small-world` und wird auf folgende Weise benutzt:

```
to build-city
   nw:generate-small-world houses links 7 7 5 false [
]
end
```

Dieser Befehl ist relativ komplex und braucht deswegen viele Argumente: als erstes, welcher Typ von Agenten benutzt werden soll (in unserem Fall houses), als zweites welche Art von Verbindungen verwendet werden soll (in unserem Fall normale "links"), dann die Zahl der Agenten (in unserem Fall 7 mal 7), dann der sogenannte Clustering-Exponent (bei uns 5) und dann entweder true oder false, je nachdem ob wir ein toroidales Netzwerk haben möchten oder nicht (in unserem Fall false). Ähnlich wie beim Befehl create-turtles können wir auch hier Details der Agenten ändern:

```
to build-city
   nw:generate-small-world houses links 7 7 5 false [
      set shape "house"
      set color red
      setxy random-xcor random-ycor
   ]
end
```

Die Farben werden wir benutzen um die verschiedenen Heiztechnologien zu kennzeichnen: Alle Häuser starten rot, da sie noch Ölheizungen verwenden. Später wird es auch Pellets-Heizungen geben (grün) und Wärmepumpen (blau). Fürs erste sollen aber alle Häuser rot bleiben. Wenn wir das Programm nun testen entsteht ein Durcheinander aus Agenten und Verbindungen. Um diese Durcheinander zu sortieren, gibt es den Befehl

```
repeat 3000 [ layout-spring houses links 0.5 5 2 ]
```

den wir am Ende des build-city Prozesses einbauen. Damit wird unser Netzwerk aus Häusern gleich aufgeräumter.

Die Stadt ist nun fertig, es passiert aber noch nichts. Wir müssen auch an der go Prozedur arbeiten. Was wir am Ende haben wollen, ist dass sich verschiedene Heiztechnologien in der Stadt ausbreiten. Momentan sind alle Häuser rot, haben also Ölheizungen. Damit aber etwas diffundieren kann, müssen wir einige Häuser mit alternativen Heizformen ausstatten. Wir erweitern unsere setup Prozedur:

```
ask one-of houses with [color = red] [
   set color green ]
ask one-of houses with [color = red] [
   set color blue ]
```

Wir wählen zufällige 2 Agenten aus, die noch mit Öl heizen und geben ihnen alternative Technologien. Wenn wir die Simulation nun starten, wird je 1 Haus grün und 1 Haus blau, aber die Innovation breitet sich noch nicht aus. Diesen Diffusionsprozess müssen wir noch in die go Prozedur schreiben.

```
to go
   diffusion
   tick
end
```

Die Prozedur diffusion muss auch erst erstellt werden. In jedem Zeitschritt soll es eine gewisse Chance geben, dass jemand mit Ölheizung auf eine Alternative eines Nachbarn umsteigt. Diese Chance nehmen wir als 1% an. Dann definieren wir den Diffusionsprozess:

```
to diffusion
   ask houses with [color = green][
      ask link-neighbors with [color = red] [
         if random 100 < 1 [
            set color green
         ]
      ]
   ]
   ask houses with [color = blue][
      ask link-neighbors with [color = red] [
         if random 100 < 1 [
            set color blue
         ]
      ]
   ]
end
```

Alle Agenten mit alternativen Heizformen werden angesprochen. Diese sprechen wiederum alle ihren Link-Nachbarn an, die noch mit Öl heizen. Man beachten den Unterschied zwischen Neighbors und Link-Neigbors: Neighbors sind alle Agenten, die auf einem benachbarten Patch stehen, Link-Neighbors sind alle Agenten, die mit einem Link (also

über das Netzwerk) mit dem Agenten verbunden sind, unabhängig von der Position. Diese Link-Neighbors haben dann eine Chance, dass sie eine neue Technologie übernehmen. Dieser einfache Prozess ist schon ausreichend um Diffusion gut abzubilden. Wenn wir die Simulation laufen lassen breiten sich die neuen Technologien aus. Am besten beobachten wir das in einem Plot. Er soll die einzelnen Technologien darstellen:

```
plot count houses with [color = green]
plot count houses with [color = blue]
plot count houses with [color = red]
```

Schön wäre auch ein abschließender Bericht, wenn die Diffusion abgeschlossen ist, also wenn niemand mehr Ölheizungen benutzt. Erstellen wir dazu eine kurze Prozedur und rufen sie in der go Prozedur auf:

AUFGABE 5.1

Wenn es keine roten Häuser mehr gibt soll die Simulation mit `stop` beendet werden. Vor dem `stop` soll auch noch die Prozedur `endreport` aufgerufen werden, die fürs erste aber leer definiert werden soll. (`to endreprort end`)

Der Bericht schreibt das Endresultat mit type und print auf den Bildschirm:

```
to endreport
   type "Diffusion complete after "
   type ticks print "ticks."
   print "Final adopters: "
   type count houses with [color = green]
   type "pellets and "
   type count houses with [color = blue]
   print "heat pump."
end
```

Sorgen wir nun noch dafür, dass das Netzwerk besser verlinkt wird. Wir erstellen einen Schieberegler mit dem Namen bonusconnections und fügen einen weiteren Teil zu build-city hinzu:

```
repeat bonus-connections [
   ask one-of houses [
   create-link-with one-of other houses with [not link-neighbor? myself] [
         set color red]
      ]
   ]
```

Dadurch werden zufällige Agenten ausgesucht, die dann einen Link zu einem anderen Agenten aufbauen, der noch nicht mit ihnen verbunden ist. Damit wir diese Verbindungen von den anderen Verbindungen unterscheiden können, färben wir sie rot ein. Mit mehr Verbindungen sollte der Diffusionsprozess nun auch schneller ablaufen.

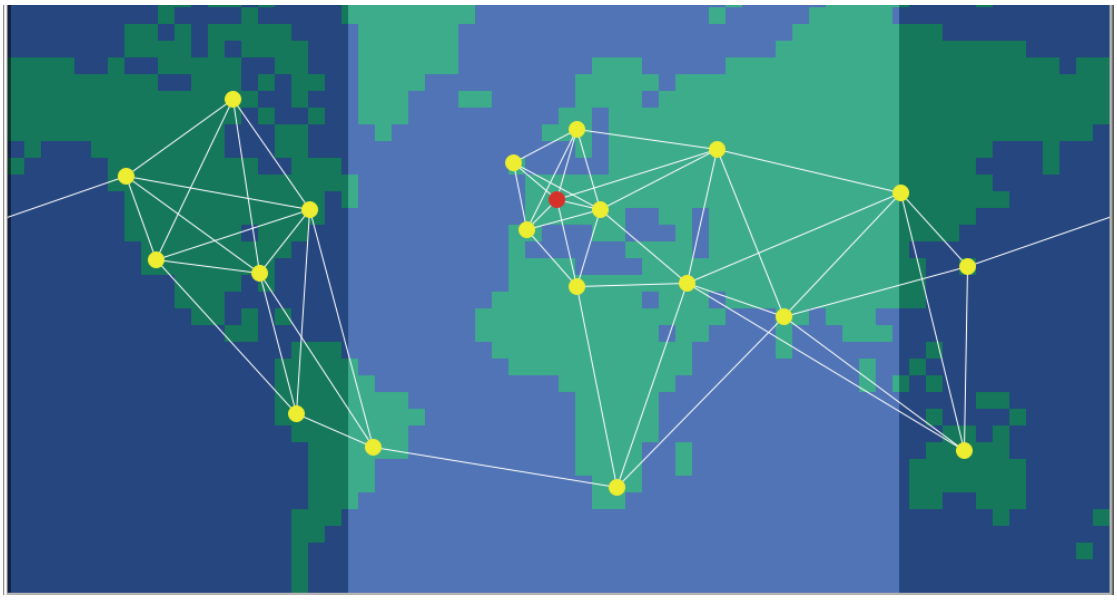
Zusammenfassung

- Für erweiterte Netzwerkbefehle braucht man die die extension nw.
- Der Befehl `nw:generate-small-world` erzeugt ein small-world Netzwerk aus Agenten.
- Das Layout eines Netzwerk optimiert man mit `repeat 3000 [ layout-spring houses links 0.5 5 2 ]`
- Agenten, die mit anderen Agenten über das Netzwerk verknüpft sind werden `link-neighbors` genannt
- Um Text auf den Bildschirm zu schreiben kann man `print`(fängt eine neue Zeile an) und `type` (fängt keine neue Zeile an) verwenden

Expert Knowledge

Die Netlogo-extension nw ist noch deutlich vielseitiger als in diesem Beispiel sichtbar wird. Es gibt viele Netzwerkgeneratoren und auch das bestimmen von vielen Netzwerkeigenschaften ist genau so möglich wie community detection. Die offizielle Beschreibung dieser extension ist eine gute Anlaufstelle: `https://ccl.northwestern.edu/netlogo/docs/nw.html`.
Hier begegnen wir auch zum ersten mal dem Wort `myself`. Dieser Befehl ist deutlich mächtiger als es hier den Anschein hat. Mit ihm können wir in den übergeordneten Kontext wechseln. Wenn also Agent A einen Agenten B aufruft und wir dann innerhalb des Kontextes von Agent B eine Eigenschaft von Agent A brauchen ist das mit `myself` möglich. Auch die Kombination `ask myself` ist erlaubt, mit der wir wieder einen Kontext weiter nach oben wechseln können. In komplizierteren Netlogo-Programmen ist das Navigieren durch die unterschiedlichen Kontexte eine große Herausforderung und `myself` ausgesprochen wichtig.

6 Modell 6 - Energienetzwerk



In diesem Modell betrachten wir ein globales Energienetzwerk. In weiterer Folge wollen wir analysieren, wie wichtig die Anzahl der Netzwerkverbindungen für eine gute Stromversorgung ist und wie gut natürliche Schwankungen ausgeglichen werden können. Für dieses Modell brauchen wir zusätzliche Daten, die man unter
`www.jaeger-ge.org/energiemodell-startpaket.zip` herunterladen kann.

Wir erstellen einen setup-Knopf. In der setup Prozedur möchten wir gerne, dass eine halbwegs realistische Karte der Erde erstellt wird. Da es viel zu umständlich wäre, die Patches alle einzeln grün (für Land) oder blau (für Wasser) einzufärben, gibt es in Netlogo die Möglichkeit, dass man fertige Grafiken lädt. Die Netlogowelt übernimmt dann die Farben der importierten Grafik. Der Befehl dazu lautet import-pcolors.

```
to setup
ca
import-pcolors "pixelmap.png"
reset-ticks
end
```

Dazu ist es wichtig, dass die Datei pixelmap.png im gleichen Ordner liegt wie unser Netlogomodell. Wenn wir den Setupknopf nun benutzen, sehen wir, dass das Bild erfolgreich importiert wird. Unsere Welt ist aber noch sehr verschwommen und undeutlich. Das liegt daran, dass das Bild, das wir einlesen, nicht das gleiche Seitenverhältnis hat wie unsere Netlogowelt. Wir müssen die Größe der Netlogowelt also anpassen. Mittels

Rechtsklick auf die Welt und Auswahl des Eintrags Edit erhalten wir einen Dialog in dem wir unsere Welt konfigurieren können. Als Koordinatenursprung wählen wir Corner und Bottom Left und die Anzahl der Patches geben wir bei max-pxcor mit 329 und bei max-pycor mit 219 an. Da wir jetzt eine sehr große Welt mit 330 x 220 Patches haben werden wir weiters noch die dargestellte Größe der Patches unter Patch size auf 3 ändern. Wir bestätigen den Dialog und unsere Welt sollte jetzt viel größer sein, aber wir sollten ungefähr die Netlogo Welt auf einmal erblicken können.

Soviel zum Einlesen der Karte. Der nächste wichtige Teil unserer Simulation werden die Produzenten und Konsumenten der Energie sein. Wir werden nun aber nicht alle Kraftwerke bzw. Städte auf der ganzen Welt einbauen, sondern nur exemplarisch einige Ballungszentren. Dennoch wird es zu viele Agenten geben, als dass wir alle per Hand eingeben können. Auch hier müssen wir also Daten von einer externen Quelle einlesen. Glücklicherweise haben wir eine fertige Tabelle mit wichtigen Ballungszentren. In dieser Tabelle steht einerseits die Position des Ballungszentrums, andererseits auch wie viel Strom hier maximal produziert, und wie viel Strom hier maximal verbraucht wird.

Die Tabelle ist im allgemeinen Format „csv“ gespeichert. Das steht für comma-separated Values, also Werte, die mit Beistrichen voneinander getrennt sind. Die Einträge sind also so aufgebaut

posx,posy, verbrauchterstrom, produzierterstrom

also beispielsweise

12,15,125,143,1

Durch diesen einfachen Aufbau, kann Netlogo die Tabelle sehr einfach lesen. Das können wir gleich austesten. Damit Netlogo csv-Dateien verwenden kann, benötigen wir die Erweiterung csv:

```
extensions [csv]
```

Damit haben wir Zugang zu einigen nützlichen Befehlen, die wir im Weiteren brauchen werden. Im ersten Schritt erstellen wir die Prozedur „make-cities“ die später einmal die Städte in unsere Welt platzieren soll, fürs erste aber nur die csv-Datei öffnet, und den Inhalt auf den Bildschirm schreibt:

```
to make-cities
file-open "cities.csv"
let row csv:from-row file-read-line
while [ not file-at-end? ] [
   show row
   set row (csv:from-row file-read-line)
]
file-close
end
```

Zuerst wird die Tabelle mit `file-open "cities.csv"` geöffnet. Der Befehl `let row csv:from-row file-read-line` liest eine Zeile aus der Tabelle ein und speichert sie unter dem Namen row. Als nächstes kommt eine while-Schleife, die so lange läuft, bis wir am Ende des files sind. Innerhalb dieser Schleife, wird die aktuelle Zeile auf den

Bildschirm geschrieben. Anschließen wird die nächste Zeile aus dem file genommen und auf die Variable row geschrieben.

Wenn wir make-cities nun aufrufen, wird der Inhalt der ganzen Tabelle Zeile für Zeile ausgegeben. Somit haben wir sichergestellt, dass die Informationen ankommen. Jetzt müssen wir sie nur noch richtig benutzen. Dazu müssen wir erst lernen, wie wir auf einzelne Zahlen innerhalb dieser Reihen zugreifen können. Innerhalb von Netlogo, sind diese Reihen als Listen gespeichert, jede Zeile ist also für Netlogo eine Liste, in der 4 Zahlen stehen. Wir wollen nun nie auf die ganze Liste zugreifen, sondern immer nur auf einzelne Einträge, in dieser Liste, denn die x-Koordinate der Stadt ist eben nicht die ganze Zeile, sondern nur die erste Zahl, die in der Liste steht. Einzelne Elemente aus Listen, bekommt man in Netlogo mit `item`. `item 0 namederliste` liefert also beispielsweise das erste Element (das mit der Nummer 0) der Liste. Diesen Befehl werden wir nun nutzen, um die relevanten Daten aus den Tabellenreihen zu extrahieren und unter einem sinnvollen Namen zu speichern:

```
to make-cities
file-open "cities.csv"
let row csv:from-row file-read-line
while [ not file-at-end? ] [
   let posx item 0 row
   let posy item 1 row
   let cons item 2 row
   let prod item 3 row
   set row (csv:from-row file-read-line)
]
file-close
end
```

Hier geben wir die Daten nun nicht mehr einfach auf den Bildschirm, sondern erstellen mit let neue Variablen, in denen die Zahlen gespeichert werden. Das macht es einfacher, neue Agenten mit genau diesen Werten zu erstellen. Dazu legen wir neue Agenten-Variablen an, damit wir den produzieren Strom und den benötigten Strom auch für jeden Agenten einzeln speichern können. Den breed nennen wir cities.

```
breed [cities city]
cities-own [production consumption]
```

Nun können wir innerhalb von make-cities auch wirklich Agenten erstellen, genauso, wie es in der csv-Tabelle steht:

```
to make-cities
file-open "cities.csv"
let row csv:from-row file-read-line
while [ not file-at-end? ] [
   let posx item 0 row
   let posy item 1 row
   let cons item 2 row
   let prod item 3 row
   create-turtles 1[
      set shape "circle"
      set size 5
      set color red
      setxy posx posy
      set production prod
      set consumption cons
   ]
   set row (csv:from-row file-read-line)
]
file-close
end
```

Wenn wir nun auf Setup drücken, werden die Agenten korrekt erstellt und durch rote Punkte dargestellt. Möchten wir später irgendwann einmal neue Agenten einbauen, reicht es, wenn wir Einträge in die csv-Tabelle anhängen.

Jede Stadt weiß nun also, wie viel Storm sie produziert und wie viel Strom sie benötigt. Um Bedarfsunterschiede auszugleichen ist es nun notwendig, dass die einzelnen Städte untereinander vernetzt sind und somit Strom handeln können. Wir erstellen ein ganz einfaches Netzwerk:

```
to connect-cities
ask cities[
   create-links-with min-n-of 4 other cities [distance myself] [
   set color white]
]
end
```

Dieser Befehl sieht recht kompliziert aus, ist aber einfach zu verstehen, wenn man ihn Schritt für Schritt analysiert: Wir sagen allen Agenten, dass sie Links erstellen sollen. Mit wem sie links erstellen sollen steht hier: `min-n-of 4 other cities [distance myself]`. Also von allen anderen Agenten (`other cities`) sollten die 4 ausgesucht werden, bei denen eine gewisse Eigenschaft am kleinsten ist (`min-n-of 4`). Und die Eigenschaft, um die es geht ist der Abstand zum ursprünglichen Agenten: (`[distance myself]`)

Nun haben wir ein sehr einfaches Netzwerk, das aber für unsere Zwecke ausreichen sollte. Wir werden in weiterer Folge analysieren, wie viel Energie gehandelt werden muss, damit alle Städte immer gut versorgt sind. Dazu benötigen wir eine go-Prozedur, die zu jeder Stunde überprüft, ob genug Strom vorhanden ist. Haben wir genug Strom, so färben wir die Stadt gelb, wenn nicht, dann färben wir sie rot. Sowohl Produktion, also auch Bedarf an Strom sind nicht immer exakt gleich, sondern mit einer Standardabweichung von 10% normalverteilt. Um das Nettoergebnis zu speichern, bekommen die Agenten auch eine neue Variable namens netto.

```
to check-energy
ask cities[
set netto production * random-normal 1 0.1 - consumption * random-normal 1
0.1
set color yellow
if netto < 0 [
   set color red
]
]
end
```

Um einen Überblick darüber zu bekommen, wie viele Städte zu wenig Strom haben, erstellen wir einen Plot, der die Anzahl der roten und grünen cities zählt. Man sieht deutlich, dass nicht alle Städte immer genügend Strom zur Verfügung haben. Wir müssen also die Möglichkeit schaffen, dass Strom gehandelt wird. Das machen wir in einer eigenen Prozedur namens trade:

```
to trade
repeat 50 [
   ask cities with [color = red][
      if any? link-neighbors with [netto > 5][
         set netto netto + 5
         ask one-of link-neighbors with [netto > 5][
            set netto netto - 5
         ]
      ]
      if netto > 0 [
         set color yellow
      ]
   ]
]
end
```

Hier wird also mit Energiepaketen der Größe 5 gehandelt: Jeden Tick wird 50 mal überprüft, ob jemand noch Energiebedarf hat, bzw. ob einer der Link-Nachbarn genug Überschuss hat, um diesen Bedarf zu decken. Diese einfache Modell kann nun benutzt

werden um verschiedene Netzwerke und verschiedene Bedarfsschwankungen zu untersuchen. Wir könnten zum Beispiel einen Schieberegler connectivity anlegen, mit dem eingestellt werden kann, mit wie vielen Städten jede Stadt verbunden ist.

AUFGABE 6.1

Die Anzahl der Verbindungen pro Agent soll mit einem Schieberegler einstellbar sein.

Man könnte es auch erweitern, um den Unterschied zwischen Tag und Nacht einzubauen. Aber wann ist auf der Erde Tag und wann Nacht? Mehr dazu unter Expert Knowledge.

Zusammenfassung

- Um Pixelgrafiken zu importieren, damit man die Farben als Patchfarben verwenden kann gibt es den Befehl `import-pcolors`.
- Numerische Daten (aber auch Zeichenketten) importiert man am besten mit der csv-extension.
- Um aus einer Gruppe von Agenten, z.B. die 7 mit dem kleinsten Wert einer gewissen Eigenschaft auszuwählen, schreibt man `min-n-of 7 agentenmenge [eigenschaft]`
- Der Befehl `distance myself` liefert den Abstand zwischen einem Agenten und dem Agenten, der ihn aufgerufen hat.

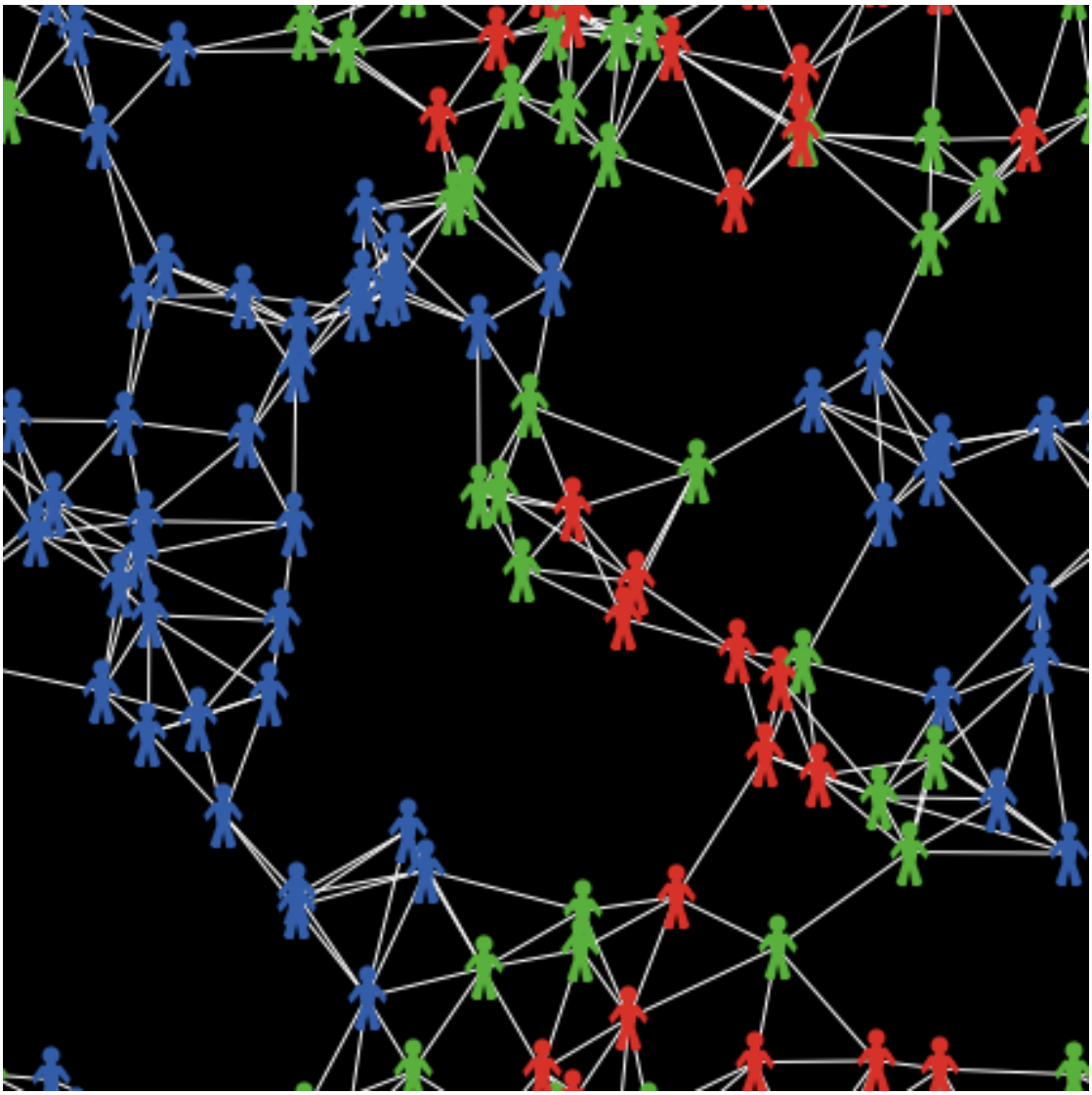
Expert Knowledge

Tag und Nacht in einem solchen Modell einzubauen ist nicht ganz so einfach, denn es ist nicht überall gleichzeitig Tag und Nacht. Man kann es aber so lösen: Man färbt die Hälfte der Karte ein wenig heller, die andere Hälfte ein wenig dunkler. Dann definiert man Linien, an denen die Sonne gerade auf bzw. untergeht. Jeden Tick wandert die Sonne dann weiter. Hier die wichtigsten Änderungen, die dazu notwendig wären:

```
globals [dayline nightline]
to setupdaynight
   ask patches with [pxcor > 165][
   set pcolor pcolor + 1
   ]
   ask patches with [pxcor <= 165][
   set pcolor pcolor - 1
   ]
   set dayline 165
```

```
   set nightline 0
end
to daynight
   ask patches with [pxcor = dayline][
   set pcolor pcolor + 2
   ]
   ask patches with [pxcor = nightline][
   set pcolor pcolor - 2
   ]
   set dayline dayline - 1
   set nightline nightline - 1
   if dayline = -1 [
   set dayline 329]
   if nightline = -1 [
   set nightline 329]
end
(...)
   if pcolor = 75.7 [
   set netto (production * random-normal 1 0.1) -
   (consumption * random-normal 1 0.1)
   ]
   if pcolor = 73.7 [
   set netto (production * random-normal 1 0.1) -
   (0.5 * consumption * random-normal 1 0.1)
   ]
```

7 Modell 7 - Rumor-Netzwerk



In Modell betrachten wir die Ausbreitung eines Gerüchts. Das Gerücht wird eine gewisse Glaubhaftigkeit haben, die eine entscheidende Rolle darin spielen wird, ob es sich über das gesamte Netzwerk verteilt, oder nicht. Mit Hilfe von Behaviorspace werden wir diesen Parameter genau analysieren.

AUFGABE 7.1

In einer Setup-Prozedur sollen 100 Agenten vom breed persons mit der shape "personërstellt werden und zufällig in der Netlogowelt angeordnet werden. Ihre Farbe soll blau sein, ihre Größe (size) 2.

AUFGABE 7.2

Jeder Agent soll sich im setup mit den 5 Agenten über ein Netzwerk verbinden, die ihm am nächsten sind. Danach soll einer der Agenten rot eingefärbt werden.

AUFGABE 7.3

Es soll einen Schieberegler mit dem Namen chance geben, der angibt wie hoch die Wahrscheinlichkeit ist, dass ein Gerücht geglaubt wird. Innerhalb der go-Prozedur sollen alle roten Agenten ihre blauen Nachbarn ansprechen. Dort wird dann eine Zufallschance entscheiden ob das Gerücht geglaubt wird (Agent färbt sich auch rot) oder nicht (Agent grün und somit immun gegen das Gerücht). Dazu benutzt man am besten eine If-Else Abfrage.

If-Else Abfragen braucht man immer dann, wenn auch etwas passieren soll, nur wenn die If-Bedingung nicht erfüllt ist. Der Aufbau lautet

`ifelse abfrage [befehle_wenn_wahr][befehle_wenn_falsch]`.

Man beachte die beiden [ ]-Blöcke.

AUFGABE 7.3

Die Simulation soll beendet werden, wenn es keine blauen Agenten mehr gibt, die rote Nachbarn haben. Hierbei geht man am besten so vor: Zuerst erstellt man eine lokale Variable und setzt sie auf 1 mit `let finished 1`. Dann fragt man, ob es überhaupt noch blaue Agenten gibt. Wenn dem so ist, frag man alle Agenten ob sie rote Link-Nachbarn haben. Sobald ein blauer Agent mit rotem Nachbarn gefunden wurde, wird finished auf 0 gesetzt, die Simulation ist dann nämlich noch nicht fertig. Am Schluss beendet man die Simulation nur dann, wenn finished auch nach dem Überprüfen noch 1 ist.

AUFGABE 7.4

Mit Behaviorspace sollen verschiedene Werte von chance durchprobiert werden. Das gesuchte Ergebnis ist die Anzahl der roten Agenten und soll nur am Ende der Simulation gemessen werden. Durch einen Plot sollte man einen interessanten Übergang sehen.

Zusammenfassung

- If-Else Abfragen sind erweiterte If-Abfragen, die auch einen Befehl ausführen, wenn die Bedingung nicht erfüllt ist. Man schreibt Sie als
  `ifelse abfrage [befehle_wenn_wahr][befehle_wenn_falsch]`.
- Wenn man Behaviorspace ohne tick-Limit verwendet, muss das Programm eine stop-Bedingung haben, die garantiert in jeder Simulation eintritt. Sonst baut man eine Endlosschleife.
- Schwankungen in Behaviorspaceanalysen kann man reduzieren, indem man die einzelnen Durchläufe wiederholt (repetitions).

Expert Knowledge

If-Else Abfragen sind ncht nur eleganter als zwei aufeinanderfolgende If-Abfragen, es gibt auch einen wichtigen Bedeutungsunterschied. Wenn man zum Beispiel alle roten Patches grün färben möchte und alle grünen patches rot, geht das mit

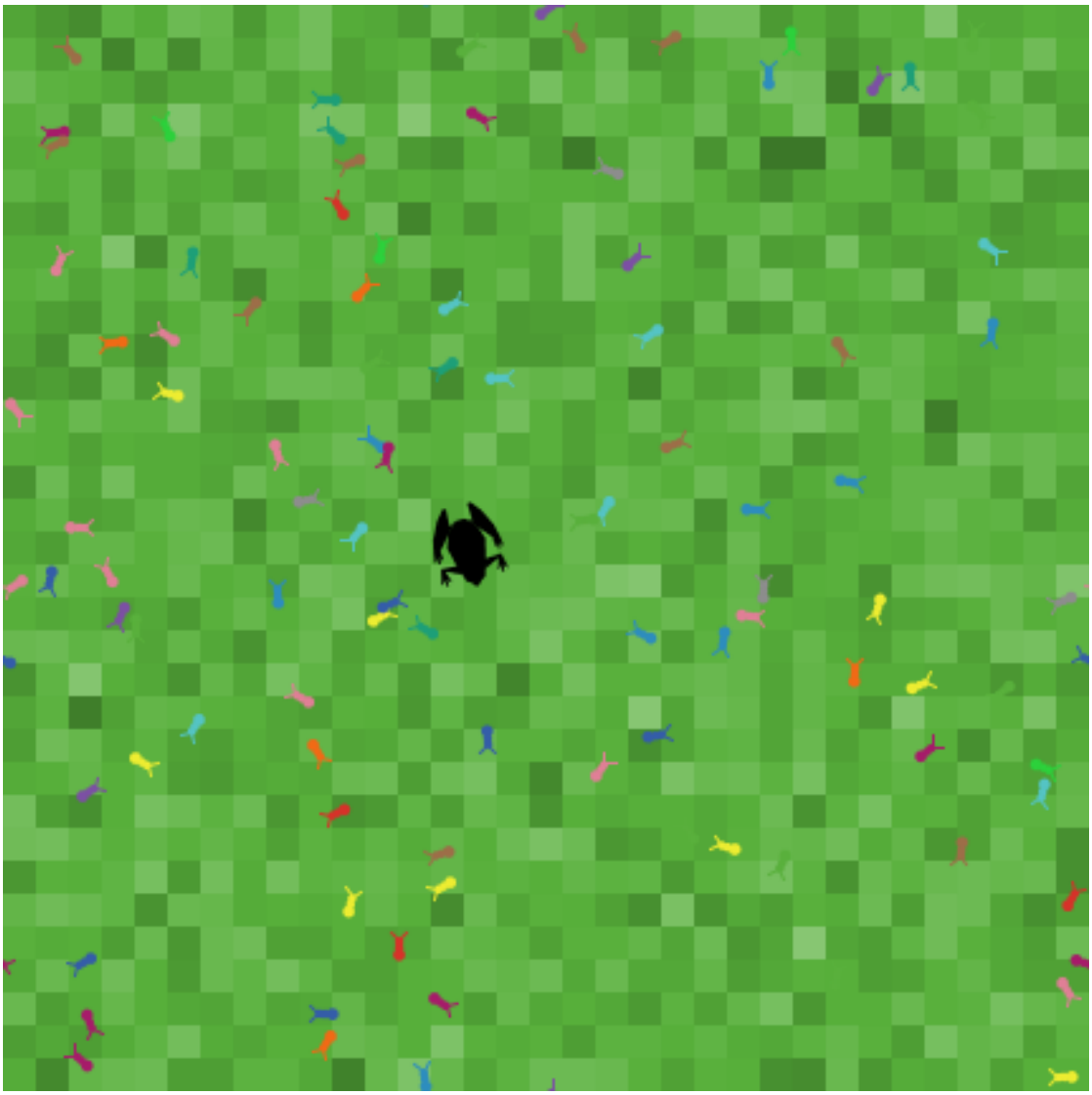
```
ask patches with [color = red or color = green][
   ifelse color = green
   [set color red]
   [set color green]
]
```

Zwei If-Abfragen hintereinander funktionieren hier aber nicht:

```
ask patches with [color = red or color = green][
   if color = green
   [set color red]
   if color = red
   [set color green]
]
```

Hier werden nämlich alle Patches grün. Warum? Patches, die am Anfang rot sind, werden in der zweiten Abfrage grün. Patches die am Anfang grün sind werden in der ersten Abfrage rot und dann sofort in der zweiten wider grün. Bei zwei If-Abfragen aknn es also sein, dass beide Befehle ausgeführt werden. Das kann mit ifelse verhindert werden.

8 Modell 8 - Evolution



In diesem Modell beschäftigen wir uns mit dem Thema Evolution. Sobald es in einem System irgendeine Form von Selektion gibt und auch die Möglichkeit zu Mutation besteht, führt das immer zu Evolution. Das sieht man auch schon in einem ganz einfachen Modell, indem ein Frosch Käfer jagt, die sich optisch gut vom Boden abheben. Später können wir auch untersuchen, was mit den gut angepassten Individuen passiert wenn sich die Umwelt schlagartig ändert (Klimawandel).

Um diese Fragen zu beantworten, beginnen wir mit einem sehr einfachen Modell. Wir erstellen einen setup und einen go Knopf und lassen im Setup-Prozess die gewünschte

Anzahl an Agenten entstehen.

```
breed [bugs bug]
breed [frogs frog]
to setup
ca
create-bugs 100[
   set shape "bug"
   set xcor random-xcor
   set ycor random-ycor
]
create-frogs 1[
   set shape "frog top"
   setcolor black
   set xcor random-xcor
   set ycor random-ycor
]
reset-ticks
end
```

Auch den Hintergrund müssen wir im setup einfärben. Das hat hier nicht nur ästhetische Gründe, sondern ist auch für das Modell wichtig: Käfer die eine ähnliche Farbe haben wie der patch auf dem sie stehen sind besser getarnt.

```
ask patches[set pcolor green + random-normal 0 0.5 ]
```

Als nächstes erstellen wir eine move-prozedur, die im Go aufgerufen werden soll.

AUFGABE 8.1

Eine Move-Prozedur soll erstellt werden. Käfer sollen sich zufällig bis zu 30 Grad nach rechts und dann bis zu 30 Grad nach links drehen. Im Anschluss gehen sie 0.1 Patches vorwärts. Der Frosch springt immer 0.5 Patches, ohne sich zu drehen.

Trifft der Frosch auf eien Käfer, so wird der Käfer gefressen. Auch das können wir in die Move-Prozedur einbauen, direkt nach dem Forwärtsbewegen.

```
if any? bugs-here [
ask bugs-here [die]
]
```

Als nächstes bauen wir die Vermehrung der Käfer ein. Wir gehen von einer konstanten Population aus, das heißt immer wenn es weniger als 100 Käfer gibt, werden so lange neue geboren, bis es wieder 100 gibt. Für so etwas können wir eine `while`-Schleife benutzen, die einen Befehl so lange ausführt, bis eine gewisse Bedingung nicht mehr erfüllt ist.

```
to reproduce
while [count bugs < 100][
ask one-of bugs [
hatch 1[
]
]
]
end
```

Nun möchten wir auch noch Mutation einbauen, also den Umstand, dass die Nachkommen nicht exakt gleich sind wie die Eltern. Wir ändern dazu die Eigenschaft `color` ein wenig. Wie viel die Änderung betragen kann legen wir mit einem Schieberegler namens `mutation-rate` fest, der von bis 5 in 0.5er-Schritten läuft.

AUFGABE 8.2

Die neugeborenen Käfer sollen eine leicht geänderte Farbe haben. Die Änderung soll normalverteilt sein mit einem Mittelwert von 0 und einer Standardabweichung von `mutation-rate`.

Um zielgerichtete Evolution zu erhalten, müssen wir auch noch eine Form der Auslese finden, den bis jetzt werden die Käfer mehr oder weniger zufällig gefressen. Dazu definieren wir eine neue Agenten-Variable namens `visibility`. Diese berechnen wir in jedem Zeitschritt neu: Es ist immer der Absolutbetrag der Differenz von der Farbe des Käfers und der Farbe des Patches auf dem er steht. Ein grüner Käfer (Farbe 55) auf grünem Grund (Farbe 55) ist also perfekt getarnt (abs(50-50) = 0). Ein dunkelgrüner (Farbe 53) ist gleich gut zu sehen wie ein hellgrüner (Farbe 57).

```
bugs-own[visibility]
(...)
to update-visibility
ask bugs [
   set visibility abs (color - pcolor)
]
end
```

Nun ändern wir noch das Verhalten des Frosches: Er soll alle Käfer in einem Radius von 10 Patches wahrnehmen können und sich dann immer zu dem drehen, der am besten sichtbar ist. Fr dieses Unterfangen brauchen wir 3 neue Netlogobefehle. Zuerst müssen wir Agenten in einem gewissen Radius um den aktiven Agenten auswählen können. Dafür gibt es den Befehl `in-radius`. Um sich zu einem gewissen Agenten zu drehen benutzen wir `face`. Das schwierigste wird sein, den Agenten auszuwählen, der am besten sichtbar ist. Dazu können wir den Befehl `max-of agentenset [eigenschaft]` benutzen, der aus einem gegebenen Agentenset, denjenigen aussucht, der den größten Wert bezüglich der

vorgegebenen Eigenschaft hat. Alle drei Befehle zusammen, drehen den Frosch dann in Richtung seines Opfers:

```
if any? bugs in-radius 10[
face max-one-of bugs in-radius 10 [visibility]
]
```

Wenn wir nun einen Plot einbauen, der die durchschnittliche Sichtbarkeit der Käfer anzeigt, sehen wir, dass sie rapide abnimmt. Auch in der Netlogowelt selbst sehen wir, dass die meisten Käfer nach einiger Zeit einen Grünton angenommen haben. Die Mutationsrate ändert wie schnell diese Anpassung geht, bzw. wie genau sie möglich ist.

Zusammenfassung

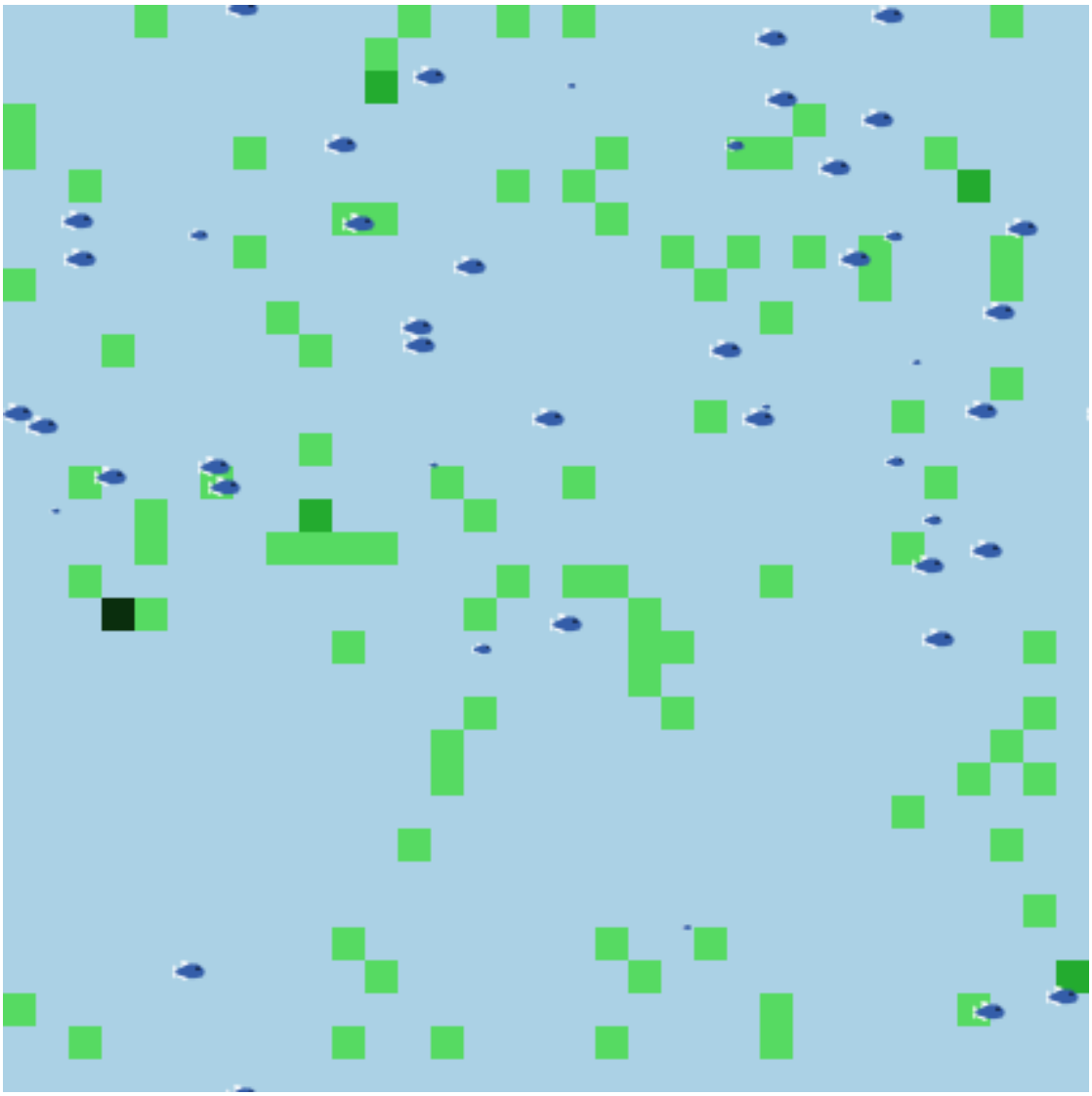
- Der Befehl `in-radius n` wählt Objekte in einem Umkreis von n Patches aus
- `face` dreht einen Agenten zum gewünschten Ziel
- mit dem Befehl `max-of agentenset [eigenschaft]` kann man aus einem gegebenen Agentenset, denjenigen aussuchen, der den größten Wert bezüglich der vorgegebenen Eigenschaft hat.
- `while` führt einen Befehl so lange aus, bis eine gewissen Bedingung nicht mehr erfüllt ist (Auchtung: Es besteht die Gefahr Endlosschleifen zu erstellen)

Expert Knowledge

Noch spannender wird das Modell, wenn wir einen Knopf hinzufügen, der die Farbe der Umgebung ändert. Vor allem wenn die Käfer schon sehr gut angepasst sind, dauert es dann viel länger um sich an die neue Umgebung anzupassen als es beim ersten mal der Fall war.

```
to environmental-change
let basecolor one-of [green brown gray yellow]
ask patches [
set pcolor basecolor + random-normal 0 0.5
]
end
```

9 Modell 9 - Populationsdynamik



In diesem Modell betrachten wir ein einfaches Populationsmodell am Beispiel von Fischen und die Auswirkungen von Giftstoffen auf ein Ökosystem. Wir beginnen mit einem setup und einen go Knopf und erstellen auch gleich einen Schieberegler mit dem Namen startpopulation. In der Setup-prozedur werden die patches gefärbt und die Agenten erstellt. Go ruft fürs erste nur einen move-Befehl auf, der die Agenten zufällig bewegt.

```
breed [fishes fish]

to setup
ca
ask patches [set pcolor 98]
create-fishes startpopulation [
set color blue
set shape "fish"
setxy random-xcor random-ycor
]
reset-ticks
end

to go
move
tick
end

to move
ask fishes[
right random 20
left random 20
fd 0.1
]
end
```

Im nächsten Schritt definieren wir das Alter der Fische. Als grobe Richtlinie: Forellen werden etwa 7 Jahre alt und sind nach 2 Jahren geschlechtsreif. Wir also eine neue Agenten-Variable mit Namen age. Die Startpopulation besitzt zufälliges Alter, jeden Tick werden alle Agenten einen Tag älter. Aber einem Alter von 7 Jahren, haben die Fische eine zufällige Chance auf natürliche Weise zu sterben.

```
fishes-own [age]
create-turtles startpopulation [
set color blue
set shape "fish"
set age 2 * 365 + random (5 * 365)
setxy random-xcor random-ycor ]

to getolder
ask fishes [
set age age + 1
if age > 7 * 356[
if random 100 < 1 [ die ] ] ]
end
```

Wenn alles korrekt funktioniert, sind die jüngsten Fische am Anfang der Simulation 2 Jahre alt. Nach 5 Jahren (1825 ticks) sollten also fast alle Fische ausgestorben sein. Das können wir schnell mit einem Plot überprüfen.

```
plot count fishes
```

Um dieses Aussterben zu verhindern, müssen wir auch die Vermehrung der Fische einbauen. In einem sehr stark vereinfachten Modell hat jeder geschlechtsreife Fisch jeden Tag eine gewisse Chance, Nachkommen zu zeugen. Diese Chance können wir mit einem weiteren Schieberegler „fertility“ regeln.

```
to newfish
ask turtles with [age > 2 * 365] [
   if random 100000 < fertility[
      hatch 1 [
         set age 0
         set size 0.3
      ]
   ]
]
end
```

Junge Fische sind kleiner als die ausgewachsenen und wachsen erst mit der Zeit zu ihrer vollen Größe. Das wird in der Prozedur getolder geregelt.

AUFGABE 9.1

Wenn ein Fisch älter ist als 365 ticks soll seine Größe auf 0.6 geändert werden. Wenn er älter ist als 2 * 365 soll seine Größe 1 sein.

Wenn wir nun die Simulation testen, merken wir, dass (je nach gewählter Fertility) die Fischpopulation entweder rasch ausstirbt, oder immer weiter exponentiell wächst. Beide Fälle sind in der Natur eher selten, denn wir haben einen wichtigen Effekt noch nicht in unsere Simulation eingebaut: Wenn der Lebensraum beschränkt ist, und schon viele Fische die Ressourcen des Ökosystems nutzen, vermehren sich die Fische langsamer. Ist viel Platz vorhanden, dann können sich die Fische besser vermehren. Wir bauen diesen Effekt ein.

```
to newfish
let effectivefertility fertility
if count turtles < 0.5 * startpopulation [
   set effectivefertility fertility * 2]
if count turtles > 2 * startpopulation [
   set effectivefertility fertility * 0.5]
ask turtles with [age > 2 * 365] [
   if random 100000 < effectivefertility[
      hatch 1 [
         set age 0
         set size 0.3
      ]
   ]
]
end
```

Nun ist die Population der Fische deutlich stabiler. Im weiteren wollen wir den Effekt von Giftstoffen im Wasser messen. Diese Giftstoffe sollen für ausgewachsene Fische harmlos sein, junge Fische können daran aber zugrunde gehen. Zuerst schreiben wir eine Prozedur, die eine gewisse Menge an Giftstoffen, die per Schieberegler gewählt werden kann, erzeugt. Die Konzentration an Giftstoffen speichern wir in einer Patchvariablen. Je nach Giftstoffkonzentration wird auch die Farbe des Patches geändert.

```
patches-own [toxinamount]
to maketoxins
repeat toxins [
ask one-of patches [set toxinamount toxinamount + 1]
]
colortoxins
end
to colortoxins
ask patches with [toxinamount = 0] [set pcolor 98]
ask patches with [toxinamount = 1] [set pcolor 66]
ask patches with [toxinamount = 2] [set pcolor 64]
ask patches with [toxinamount >= 3] [set pcolor 61]
end
```

Nun lassen wir diese Giftstoffe diffundieren. In jedem Zeitschritt wandern die Giftstoffe auf einen zufälligen benachbarten Patch.

```
to movetoxins
ask patches with [toxinamount > 0][
while [toxinamount > 0] [
set toxinamount toxinamount - 1
ask one-of neighbors [set toxinamount toxinamount + 1]
]
]
colortoxins
end
```

Die Giftstoffe sollen nun eine Auswirkung auf die nicht ausgewachsenen Fische haben. Das bauen wir am besten in die getolder Prozedur ein.

```
to getolder
ask turtles [
set age age + 1
if age > 7 * 356[
if random 100 < 1 [
die
]
]
set size 0.3
if age > 365 [ set size 0.6]
if age > 2 * 365 [ set size 1]
if size = 0.3 and random 100 < [toxinamount] of patch-here [die]
if size = 0.6 and random 1000 < [toxinamount] of patch-here [die]
]
end
```

Nun können wir für jeden Wert von Startpopulation, Giftstoffen und Vermehrungsrate, Experimente machen und mit Behaviorspace auswerten. Interessant ist auch ein Plot, der uns zeigt, wie viele Fische von welcher Größe vorhanden sind:

```
plot count turtles
plot count turtles with [size = 0.6]
plot count turtles with [size = 0.3]
```

Zusammenfassung

- Neben Agenten-Variablen gibt es auch Patch-Variablen, die mit `patches-own` definiert werden.
- Mit `repeat` können wir einen Befehl mehrmals hintereinander ausführen lassen.
- `while` führt einen Befehl so lange aus, bis eine gewissen Bedingung nicht mehr erfüllt ist (Auchtung: Es besteht die Gefahr Endlosschleifen zu erstellen)
- Um Eigenschaften des Patches auf dem Man sich befindet zu bekommen, schreibt man wie üblich die gewünschte Eigenschaft in eckigen Klammern gefolgt von `of`. Der Patch auf dem sich der Agent befindet heißt `patch-here`

Expert Knowledge

In diesem Modell sieht man unter anderem auch den Unterschied zwischen `ask n-of 100 patches [...]` und `repeat 100 [ask one-of patches[...]]` Bei der ersten Variante werden von einer gegebenen Menge (hier: alle patches) 100 ausgesucht, die dann einen Befehl bekommen. Ist die Menge nicht groß genug für n, so bekommt man eine Fehlermeldung. Kein Patch wird den Befehl 2 mal bekommen. Anders bei Variante 2: Hier wird ein zufälliger Patch ausgesucht und der Befehl gesendet. Erst nach dem Befehl wird wiederholt und ein zweiter zufälliger Patch wird ausgesucht. Ein Patch kann also mehrmals ausgesucht werden und man bekommt keine Fehlermeldung, auch wenn die Menge nur aus 1 Objekt bestehen würde.

10 Modell 10 - Ökosystem

In diesem Modell betrachten wir ein vollständiges Ökosystem, bestehend aus Pflanzen, Pflanzenfressern und Fleischfressern. Dazu benötigen wir mehrere breeds gleichzeitig, die miteinander auf verschiedene Arten interagieren, um ein stabiles Ökosystem zu erzeugen.

AUFGABE 9.1

Es soll drei verschiedene breeds geben: Pflanzen, Pflanzenfresser und Fleischfresser. Jeder breed hat einen eigenen Schieberegler, mit dem die Startpopulation eingestellt werden soll. Die Agenten sind zufällig in der Welt verteilt.

AUFGABE 9.2

Die Go-Prozedur soll auch drei Teilen bestehen: move, eat und survive. Move soll folgendermaßen aussehen: Alle Agenten, die nicht zu den Pflanzen gehören, sollen sich zufällig drehen und dann 1 patch nach vorne gehen. Der Begriff für Agenten, egal welcher breed heißt in Netlogo historisch bedingt nicht etwa agents, sondern turtles. Man kann also `ask turtles with [breed != ...]` verwenden.

AUFGABE 9.3

Wir werden die Nährstoffe jedes Agenten in seiner Größe speichern. Die Prozedur eat ist für alle breeds ein wenig anders: Die Pflanzen wachsen in jedem tick um 0.01, außer auf diesem Patch befindet sich schon eine Pflanze. `(if not any? other breedname here)`
Die Pflanzenfresser werden jeden tick durch Hunger um 0.01 kleiner. Wenn sie auf einem Patch mit einer Pflanze stehen, wird eine der Pflanzen um 0.1 kleiner, der Pflanzenfresser wird um 0.1 größer. Fleischfresser werden jeden tick durch Hunger um 0.005 kleiner. Wenn sie auf einem Patch mit einem Pflanzenfresser stehen, stirbt der Pflanzenfresser und der Fleischfresser wird um 0.5 größer.

AUFGABE 9.4

Die survive Prozedur regelt Sterben und Nachkommen. Das ist für alle breeds gleich und kann somit wieder mit `ask turtles` gemacht werden. Agenten, die kleiner sind als 0.5 sollen sterben. Agenten, die größer sind als 2 setzen ihre Größe zurück auf 1 und erstellen danach einen Nachkommen mit `hatch`. Der neue Nachkomme wird zufällig gedreht `right random 360` und geht dann sofort einen patch vorwärts (auch Pflanzen). Ein Plot soll zeigen, wie viele Agenten von welchem breed es gibt.

AUFGABE 9.5

Um zu verhindern, dass eine Spezies völlig ausstirbt kann man eine If-Abfrage einbauen: Wenn es keine Agenten mehr von einer gewissen Spezies mehr gibt, sollen wieder 2 erstellt werden.

Zusammenfassung

- Um alle Agenten, egal welcher breed, anzusprechen verwendet man in Netlogo den Befehl `turtles`.
- `turtles` funktionert auch mit anderen Befehlen: `create-turtles, turtles-here, count turtles...`
- Nach einem `hatch` Befehl kann man den erstellten Agenten nicht nur mit set-Befehlen ändern, sondern ihn auch schon einmalig bewegen.

Expert Knowledge

Netlogo bietet auch während dem das Modell läuft viele Möglichkeiten zur Interaktion. Beispielsweise kann man einbauen, dass neue Bäume dort entstehen, wo man mit der Maus klickt. Dazu braucht man die Befehle `mousedown?` `http://ccl.northwestern.edu/netlogo/docs/dict/mouse-down.html` und die Koordinaten der Maus `http://ccl.northwestern.edu/netlogo/docs/dict/mouse-cor.html`.
Außerdem könnte man neue Buttons erstellen, die jeweils eine neue Pflanze, einen neuen Pflanzenfresser, oder einen neuen Fleischfresser erzeugen. Jeder Button in Netlogo hat, wenn man ihn erstellt auch ein Feld namens action key. Wenn wir dort einen Buchstaben eingeben, wird der Knopf immer dann aktiviert, wenn diese Taste auf der Tastatur gedrückt wird. Es wäre als sehr einfach, das Modell zu einem Spiel umzubauen, bei dem es darum geht, dass keine der Spezies ausstirbt.