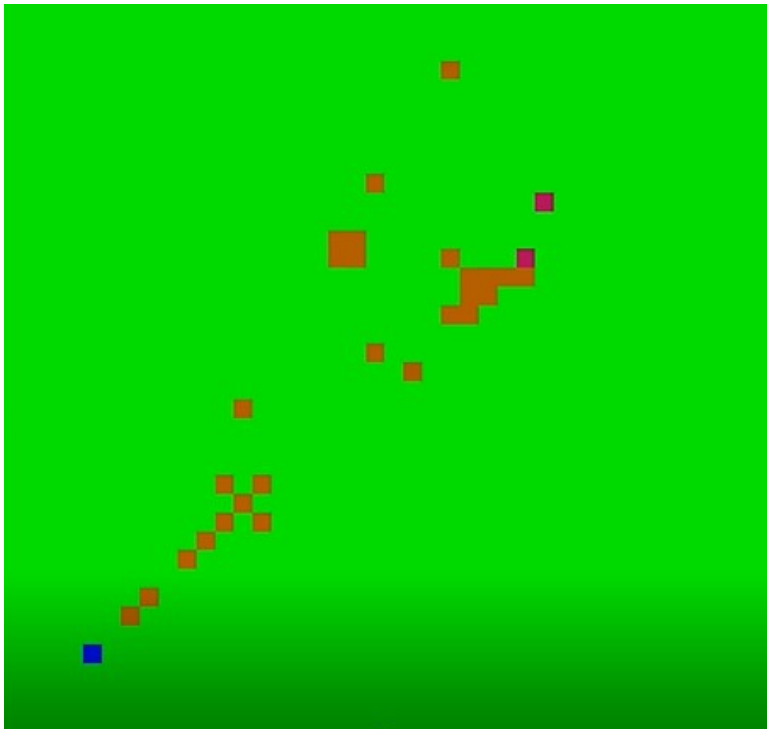
Dokumentation der Ant-Simulation

Von Jochen Stier, bei Christina Ludwig

Computer Science for geographers, Sommersemester 2019



Inhaltsverzeichnis

1 Struktur des Programms

Das Programm besteht aus sieben Modulen. Ausgeführt wird es über das „main.py“ Modul. Die main initiiert alle wichtigen Abläufe, welche in „controller.py“ näher definiert sind. Der Controller hat Zugriff auf das Spielfeld in „field.py“ und alle Spielsteine, welche in „objects.py“ definiert sind. „utils.py“ enthält Hilfsfunktionen, „errors.py“ enthält einen selbst definierten Error und „test_area.py“ führt unittests aus. Im folgenden wird auf jedes Modul näher eingegangen.

1.1 main.py – Der Mainager

Der Vergleich zwischen der main und einem Manager trifft gut den Aufgabenbereich des main-Moduls. Es gibt den Startbefehl für alle wichtigen Schritte, wirklich Ahnung, was in den unteren Ebenen passiert, hat es aber nicht. Dementsprechend wird hier auch nur controller.py und errors.py importiert. An den Controller leitet die main die Befehle weiter und errors.py versorgt sie mit Informationen über Fehlfunktionen.

Die main Funktion selbst kann in drei Abschnitte unterteilt werden. Im Ersten werden die Einstellungen geladen, sowie das Spielfeld initiiert und mit Spielfiguren besiedelt. Im Zweiten werden eine beliebige Anzahl an Runden simuliert und der Fortschritt an die Konsole ausgegeben. Zum Schluss, im dritten Abschnitt, wird eine mp4-Datei erzeugt, welche aus den Frames der einzelnen Runden aufgebaut ist und einige Kennzahlen ausgegeben, welche einen groben Überblick über den Endstand der Simulation geben.

Die main Funktion wird automatisch gestartet wenn das main-Modul ausgeführt wird. Falls die main Funktion mit dem selbst definierten „MovementError“ ended, wird sie solange erneut ausgeführt, bis das Ende der Simulation erreicht wird. Näheres zu dem MovementError steht unter Kapitel 1.6 sowie Kapitel 3.2.

1.2 controller.py – Die Kontrollräume

Die einzelnen Funktionen im controller.py könnte man, dem Vergleich aus 1.1 folgend, als das Middle Management beschreiben. Sie haben jeweils ein Aufgabenfeld, welches aus mehreren Einzelschritten besteht. Die Reihenfolge der Funktionen folgt zwecks Übersichtlichkeit grob der Verwendung in der main. Die Funktionen, welche die Erstellung des Spielfelds samt Spielfiguren koordinieren und diejenigen, welche für die Statistiken und die Erstellung der mp4 verantwortlich sind, sollten aufgrund ihrer Kürze selbsterklärend sein. Besondere Aufmerksamkeit verdienen aber die next_step und die collision_check Funktionen, die beide bei der Simulation der einzelnen Spielrunden zum Einsatz kommen. Die next_step Funktion besteht vereinfacht aus den folgenden Schritten:

1. Gewünschten Spielzug für jede Ameise berechnen und tote Ameisen löschen[1]
2. Leere Food Objekte löschen und neue mit einer 1/80tel Wahrscheinlichkeit erstellen
3. Bewegungen von Ameisen auf Kollisionen überprüfen und auf dem Spielfeld platzieren

1 Löschen des Objekts passiert in dem Code immer nur implizit durch löschen aus einer Liste, der Python GarbageCollector erledigt den Rest.

4. Wenn möglich neue Ameise auf dem Ameisenhaufen erstellen
5. Food und Hive auf dem Spielfeld platzieren
6. Fertige Version des Spielfeldes speichern

Die collision_check Funktion kommt dabei in Schritt 3 zum Einsatz. Da die Vermeidung von Kollisionen sich als komplizierter als erwartet herausgestellt hat, ist diese Funktion mit mehreren Unterfunktionen ausgestattet. Der grobe Ablauf dieser Funktion ist wie folgt:

1. Alle Ameisen auf ein zufälliges Nachbarfeld stellen, die sonst stehengeblieben wären
2. Alle Ameisen, dessen gewünschtes Feld dadurch verstellt wurde, auf ein zufälliges Nachbarfeld stellen
3. Alle Ameisen, dessen gewünschtes Feld nun noch nicht verstellt ist, bewegen
4. Alle Ameisen, die sich während Schritt 3 nicht bewegen konnten, auf ein zufälliges Nachbarfeld bewegen

Sollte es vorkommen, dass sich eine Ameise nicht bewegen kann, weil alle validen Positionen besetzt sind, dann wird der MovementError erzeugt.

1.3 field.py – Das Spielfeld

In diesem Modul liegt die Field Klasse. Eine Instanz dieser Klasse wird zu Anfang der Simulation erstellt und während der gesamten Laufzeit genutzt. Die Instanzfunktionen dienen der Kreation und Veränderung des Spielfeldes und stellen sicher, dass sein Zustand zu bestimmten Punkten valide ist. Zum Beispiel wird in der next_step Funktion zweimal kontrolliert, ob die Anzahl der aktiven Ameisen mit der Anzahl der Ameisen auf dem Spielfeld übereinstimmt.

1.4 objects.py – Die Spielfigurenkiste

Es gibt 3 verschiedene Arten von Objekten, die sich auf dem Spielfeld befinden können: Die Ameisen, das Futter und der Ameisenhaufen. All diese Objekte sind in dem Code als Klassen realisiert und befinden sich im Modul objects.py.

Die Ant Klasse bietet die meisten Funktionalitäten. Ameisen können älter werden, sterben, Essen tragen und ihre Wunschposition für den nächsten Spielzug planen. Die Funktion, mit der die Wunschposition bestimmt wird, ist der Aufgabenstellung nachempfunden. Es wird jedoch mit genormten Vektoren, nicht mit Winkeln gearbeitet, da es bei der Mittelwertberechnung der Winkelimplementation zu Problemen kam. So ist der Mittelwert von 350° und 10° gleich 180°, die Ameise sollte sich aber Richtung 0° bewegen. Eine kurze und intuitive Lösung zu diesem Problem wurde nicht gefunden, deshalb schien der Lösungsweg über Vektoren geschickter.

Die einfachste Klasse ist Food. Dieses Objekt besteht lediglich aus einer Verortung auf dem Spielfeld und der Anzahl des enthaltenen Futters (zwischen 4 und 20). Nimmt eine Ameise etwas von dem Futter, sinkt die Anzahl um 1, wenn sie 0 erreicht, wird das Objekt gelöscht. Der Hive ist ebenfalls sehr einfach aufgebaut. Hier kann Essen gelagert werden. Wenn genügen Essen

eingelagert wurde, kann damit eine neue Ameise erstellt werden. Jedoch ist dies auf maximal eine Ameise alle vier Runden begrenzt.

1.5 utils.py – Eine Prise Mathe und Hilfsfunktionen

Hier liegen die Hilfsfunktionen zur Bestimmung des nächsten Schrittes der Ameisen sowie Funktionen, welche dabei helfen, die Spielfiguren auf das Spielfeld zu bringen.

1.6 errors.py – Das rote Lämpchen

Auch zu diesem Modul lässt sich nicht viel sagen, hier ist lediglich der MovementError ohne weitere Funktionalitäten implementiert.

1.7 test_area.py – Die Sandbox

Das Testarea dient zum Testen der wichtigen Funktionen. Dies gelang mit dem Python Modul unittest. Besonders die Grundfunktionen der einzelnen Klassen, sowie die mathematischen Hilfsfunktionen wurden getestet, da sich deren korrektes Verhalten nicht unbedingt durch die erstellte mp4-Datei überprüfen lässt. Die Funktionalität der anderen Funktionen wird meist direkt im Code durch passende Asserts gewährleistet.

2 Verwendete best und good enough practices

Bei der Erstellung des Codes wurden sowohl einige der best (Wilson et al., 2014), als auch der good enough practices (Wilson et al., 2017) genutzt. Zur Übersichtlichkeit wird jeweils ein Beispiel für eine Implementation unter den jeweiligen Punkt gesetzt. Sollte ein Punkt nicht implementiert sein, wird dieser kursiv geschrieben. Praktiken, die in beiden Listen auftreten, werden nicht doppelt kommentiert.

2.1 Best practices

1. **Write programs for people, not computers.**
    a. **A program should not require its readers to hold more than a handful of facts in memory at once.**

       Kapseln von Tasks in Funktionen und Klassen limitiert Dinge die gleichzeitig im Kopf sein müssen.

    b. **Make names consistent, distinctive, and meaningful.**

       Alle Namen lassen sich intuitiv ihrer ungefähren Funktion zuordnen.

    c. **Make code style and formatting consistent.**

       Pep8 Standard immer eingehalten, zudem sehr hohe Pylint Scores im gesamten Code. Mehr zu den Stellen in denen Pylint Fehler findet in Kapitel 3.1.

2. **Let the computer do the work.**
   a. **Make the computer repeat tasks.**

      Ja.
   b. **Save recent commands in a file for re-use.**

      Kein coden in der Commandline.
   c. **Use a build tool to automate workflows.**

      Installation via pip install, falls das gemeint ist.
3. **Make incremental changes.**
   a. **Work in small steps with frequent feedback and course correction.**

      Die meisten git commits waren kleine Veränderungen, nur zu Beginn wurden sehr große Veränderungen auf einmal gemacht.
   b. **Use a version control system.**

      Git wurde von Anfang an benutzt.
   c. **Put everything that has been created manually in version control.**

      Jedes Commit enthält nur Dateien wenn diese auch verändert wurden.
4. **Don't repeat yourself (or others).**
   a. **Every piece of data must have a single authoritative representation in the system.**

      Durch Kapselung in Funktionen und Klassen fast immer erreicht.
   b. **Modularize code rather than copying and pasting.**

      Programm ist aus Modulen aufgebaut und innerhalb der Module aus Klassen und Funktionen.
   c. **Re-use code instead of rewriting it.**

      Die animation Funktion wurde beinahe komplett von der Forest_Fire Aufgabe übernommen.
5. **Plan for mistakes.**
   a. **Add assertions to programs to check their operation.**

      Während next_step wird zweimal die Konsistenz zwischen der Anzahl an Ameisen in der Liste und auf dem Spielbrett überprüft, zudem werden die Inputs ausführlich getestet.
   b. **Use an off-the-shelf unit testing library.**

      Nutzung des unittest Moduls.
   c. *Turn bugs into test cases.*

      Nicht bewusst eingesetzt, eher mithilfe von assert im Code gelöst.

   d. **Use a symbolic debugger.**

      Pycharm Debugger wurde gelegentlich verwendet.

6. **Optimize software only after it works correctly.**

   a. Use a profiler to identify bottlenecks.

      Bottleneck ist Erstellung der mp4, nicht der Rest des Codes. Dies ist leicht daran erkennbar wie viel Zeit zwischen der Ausgabe der Statistiken und dem Laufzeitende des Programmes entsteht.

   b. **Write code in the highest-level language possible.**

      Python dürfte diese Anforderung erfüllen.

7. **Document design and purpose, not mechanics.**

   a. **Document interfaces and reasons, not implementations.**

      README.md und diese Dokumentation geben einen Überblick wie das Programm funktioniert und wie es aufgebaut ist, ohne speziell auf den Code an sich einzugehen.

   b. **Refactor code in preference to explaining how it works.**

      Nur next_step und collision_check sind mit vielen Kommentaren ausgestattet, um sicher zu gehen, dass ihre Funktion klar ist.

   c. **Embed the documentation for a piece of software in that software.**

      Docstrings dürften ein Beispiel dieser Praktik sein.

8. Collaborate.

      Durch Aufgabenstellung als Soloaufgabe verboten.

Insgesamt wurden hier also nur zwei erreichbare Praktiken nicht benutzt. Einen Profiler für bottlenecks und das explizite Tests Erstellen, nachdem ein Bug gefunden wurde. Die Nutzung eines Profilers wäre vor allem bei größeren Programmen sehr sinnvoll gewesen, im Falle der ants-Simulation wäre aber der Zeitaufwand nicht mit dem potenziellen Nutzen rechtfertigbar.

2.2 Good enough Practices

1. Data management

   Da es keine Rohdaten gibt, fällt dieser Punkt komplett weg.

2. **Software**

   a. Place a brief explanatory comment at the start of every program.

   b. Decompose programs into functions. (duplicate)

   c. Be ruthless about eliminating duplication. (duplicate)

   d. Always search for well-maintained software libraries that do what you need.

      Einige komplexere Funktionen werden durch fremde Module übernommen, jedoch wurde nicht immer gesucht, ob es für eine Funktion bereits Code gibt. Beispielsweise

die mathematischen Funktionen in utils sind garantiert in einem frei verfügbaren Modul implementiert.

e. Test libraries before relying on them.

f. Give functions and variables meaningful names. (dublicate)

g. Make dependencies and requirements explicit.

Import Statements im Code, sowie requirements.txt für die im Projekt genutzten Module.

h. Do not comment and uncomment sections of code to control a program's behavior.

Wird nicht gemacht.

i. Provide a simple example or test data set

Irrelevant weil keine Roh-Daten verwendet werden.

j. Submit code to a reputable DOI-issuing repository.

Durch Aufgabenstellung verboten.

3. Collaboration

Durch Aufgabenstellung verboten.

4. Project organization

a. Put each project in its own directory, which is named after the project.

Ein Projekt, ein Ordner.

b. Put text documents associated with the project in the doc directory.

Die Datei README.md liegt im Projektordner.

c. Put raw data and metadata in a data directory and files generated during cleanup and analysis in a results directory.

ant.mp4 wird in results gespeichert.

d. Put project source code in the src directory.

Alle Module außer main.py sind im src Ordner.

e. Put external scripts or compiled programs in the bin directory.

f. Name all files to reflect their content or function. (dublicate)

5. Keeping track of changes

a. Back up (almost) everything created by a human being as soon as it is created.

Backup ist durch github realisiert. Am Ende eines Arbeitstages wurde stets commited.

b. Keep changes small. (dublicate)

c. Share changes frequently. (dublicate)

d. Create, maintain, and use a checklist for saving and sharing changes to the project.

e. Store each project in a folder that is mirrored off the researcher's working machine.

Durch github automatisch erfüllt.

f. Add a file called CHANGELOG.txt to the project's docs subfolder.

Das eheste was einem changelog ähnelt ist der git log mit den Änderungskommentaren.

g. Copy the entire project whenever a significant change has been made.

Hätte implementiert werden können, indem kleine Veränderungen in einem git branch dev hochgeladen werden und regelmäßig, nachdem insgesamt großen Veränderungen erreicht worden sind, dieser Branch mit dem Master gemerged wird.

h. Use a version control system. (duplicate)

6. Manuscripts

a. Write manuscripts using online tools with rich formatting, change tracking, and reference management.

Libre Office für die Ausarbeitung, Markdown für das readme.

b. Write the manuscript in a plain text format that permits version control.

Libre Office für die Ausarbeitung, Markdown für das readme.

Von den good enough practices wurde auch der größere Teil erreicht. Practices die leicht umsetzbar gewesen wären, waren ein expliziter Changelog und der im Kommentar unter dem Punkt 5g. beschriebene Workflow mit zwei verschiedenen Branches.

3 Bekannte Probleme

3.1 Pylint

Da der Pylint Score in die Benotung des Projektes einfließt, wurde darauf geachtet die Anforderungen von Pylint sehr genau zu befolgen. Dennoch sind im finalen Code einige Pylint Warnungen anzutreffen. Im Folgenden soll auf Einige eingegangen werden.

1. NOT_FINISHED in main.py ist eigenlich keine Konstante:

Dies ist kein Pylint Fehler im endgültigen Programm, jedoch wurde not_finished aufgrund der Pylint Warnung, man müsse Konstanten groß schreiben, zu NOT_FINISHED umbenannt. Hierbei handelt es sich aber eigentlich um gar keine Konstante, ergo ist durch die Vermeidung einer Pylint Warnung hier ein Namensfehler in den Code gekommen.

2. up nicht snake_case konform:

In einer Unterfunktion von collision_check im Modul controller.py gibt es vier Variablen für die jeweiligen Richtungen. Und zwar up, down, left und right. Eine Variable muss aber für Pylint mindestens drei Buchstaben haben. Das selbe Problem gab es auch bei der Verwendung von x und y für die Koordinatenfeldachsen.

3. counter als nicht benutzte Variable:

In create_ants, ebenfalls im controller Modul, wird eine for-Schleife benutzt, welche counter als Zählvariable beinhaltet. In der Schleife selbst erfüllt sie aber keinen Zweck. Ein Umschreiben zu einer while Schleife wäre aber unübersichtlicher. Deshalb wurde diese Warnung ignoriert.

4. Zu viele statements und branches:

Über die Instanzfunktion move in der Klasse Ant des Moduls objects.py beschwert sich Pylint besonders. Dies liegt vor allem daran, dass einige Unterfunktionen definiert wurden, welche Pylint aber in der Zählung der branches und statements ignoriert und alles der move Funktion zuordnet. Hier wäre es leicht möglich die Unterfunktionen aus der Funktion zu ziehen, jedoch vermindert dies aus subjektiver Sicht die Lesbarkeit des Codes, da die Unterfunktionen sonst ohne Kontext stehen würden.

3.2 Andere

Ein großer Bug ist bekannt, für ihn wurde aber keine einfache Lösung gefunden. Es handelt sich um die Situation, in der eine Ameise keine leeren Felder um sich herum findet und auch nicht stehen bleiben kann, weil sich in dieser Spielrunde bereits eine Ameise unter sie gestellt hat. Für diesen Fall wurde die eigene Errormeldung MovementError geschaffen. Sollte dieser Fall auftreten, startet das Programm so lange neu bis das Ende erreicht wird.

Ursprünglich war das größte Problem bei der Lösungsfindung, dass Ameisen laut Aufgabenstellung nicht stehen bleiben dürfen. Sonst wäre es ein leichtes gewesen, Ameisen nicht zu erlauben auf Felder zu gehen, die zur Zeit besetzt sind. Diese Anforderung wird mittlerweile jedoch trotzdem nicht mehr erfüllt. Es ist Ameisen nun erlaubt zu prüfen, ob sie stehen bleiben können, falls kein anderer Platzt um sie herum mehr frei ist.

Literatur

WILSON, G. ET AL. (2014): Best Practices for Scientific Computing. PLOS Biology, 12(1)

WILSON, G. ET AL. (2017): Good enough practices in scientific computing. PLOS Computational Biology, 13(6)