Agentenbasierte Simulation der COVID-19 Pandemie mit einem SEIR-Modell

Till Zemann (808255) und Ben Kampmann (803374)

Universität Potsdam
{kampmann, zemann}@uni-potsdam.de

10. Februar 2023

I Abstract

Wir simulieren den Ausbruch der COVID-19 (Coronavirus Disease 2019) Pandemie, die durch den SARS-CoV-2 Coronavirus übertragen wird. Dafür kann das Susceptible-Infected-Recovered (SIR) Modell als Differentialgleichungssystem verwendet werden. Auch eine Partikelsimulation des Susceptible-Exposed-Infected-Recovered (SEIR) Modells ist möglich, bei der man die Interaktionen einzelner Agenten (Partikel) simuliert [3]. Beide Modellvarianten haben verschiedene Vor- und Nachteile. Differentialgleichungen sind die häufigere Variante in der Praxis [4], da aufgrund der simplen Modellierung des Problems viele Konfigurationen der Parameter simuliert werden können. Zusätzlich ist der Entwicklungsaufwand deutlich niedriger verglichen mit dem agentenbasierten Modell. Dafür generiert eine agentenbasierte Simulation umfassendere Daten zur weiteren Analyse, die potentiell mehr Erkenntnisse hervorbringt. Die Anzahl der zu simulierenden Agenten befindet sich für lokale Gebiete (wenige Quadratkilometer Fläche) im umsetzbaren Rahmen. Deswegen fokussieren wir uns auf den Golmer Universitätskampus der Universität Potsdam und implementieren das SEIR-Modell als agentenbasierte Partikelsimulation, um informierte Schlüsse aus den detaillierteren Daten ableiten zu können. Der Code ist öffentlich auf unserem GitHub repository verfügbar: particle_sim.

II Zielstellung und Umfang

Ziel des Projekts ist es, die Ausbreitung des SARS-CoV-2 Coronavirus durch eine Simulation auf dem Golmer Campus der Universität Potsdam nachzubilden und dafür ein angemessenes Modell zu finden. Wir vergleichen die Eigenschaften und Nützlichkeit von zwei Modellen (SIR und SEIR), die für die Simulation verwendet werden können und präsentieren unsere Implementierung einer Partikelsimulation des SEIR-Modells mit einem integrierten Flow Field Pathfinding Algorithmus. Anschließend zeigen wir die Ergebnisse für ein Beispielexperiment mit unserem Simulator und erklären die Funktionsweise und Benutzung.

Aufbauend auf unserem Projekt könnte man verschiedene Parameter testen und weitere Features implementieren, um beispielsweise informierte Entscheidungen zur Online-Lehre und Maskenpflicht auf dem Campus zu treffen oder die Ausbreitung des Virus in einem größeren Gebiet mittels einer höheren Auflösung zu simulieren. Es gibt viele Möglichkeiten für die Integration von echten Daten, u.a. kann die Personenanzahl jedes Fachbereiches ermittelt werden und das Fachgebiet einer Person in die Wahl des Zielgebäudes eingehen (ein Physiker geht mit deutlich höherer Wahrscheinlichkeit zum Physikgebäude als ein Biologe). Weiterhin kann man den Zug verwenden (der bereits funktionsfähig ist), um Einblicke zu bekommen, wie sich das Pendeln von und zu der Universität auf die Pandemieverbreitung auswirkt.

Um den Simulator tatsächlich als Hilfsmittel für Entscheidungen zu verwenden, sollten die generierten Daten vorerst mit echten Daten der letzten beiden Jahre verglichen werden, um den Realitätsgrad und potentielle Abweichungen zu ermitteln.

III Einleitung

Eine neue Art des Coronavirus kann sich in einer Bevölkerung ausbreiten, wenn die Variante des Virus von Tieren (z.B. Fledermäusen) auf Menschen transferierbar ist [1]. Dieser Transfer wird auch zoonotische Übertragung genannt, bei der sich das zoonotische Virus, welches eine organische Struktur ohne Stoffwechsel ist, über Wirtszellen vermehren kann [2]. Die Übertragung des Virus erfolgt durch Tröpfchen und Aerosole (in die Luft verteilte Partikel), die u.a. beim Husten oder Sprechen verteilt werden. Gelangen Partikel mit dem Virus an die Schleimhäute einer gesunden Person, kann sich diese ebenfalls infizieren [5]. Dabei ist die Ansteckungsrate in Gebäuden höher, da sich die Aerosole bei unzureichender Belüftung ansammeln [5]. Dieser Umstand wird in der Partikelsimulation durch die häufigere Kollision von Agenten in Innenräumen indirekt abgedeckt. Deswegen können wir die Ansteckungswahrscheinlichkeit als konstant modellieren, ohne dass die Simulation unrealistisch wird. Wir nehmen in unserem Experiment eine Ansteckungswahrscheinlichkeit von 30% an; diese ist deutlich höher als in der Realität [6], erlaubt uns aber kürzere Simulationen durchzuführen. Die Ansteckungswahrscheinlichkeit kann in der Parameterkonfiguration geändert werden, um verschiedene Szenarien (z.B. das Tragen von Masken) zu simulieren und Strategien für die Pandemieeindämmung abzuleiten.

Wir nehmen an, dass von einer gefixten Populationsgröße N zum Startzeitpunkt ($t = 0$) der Pandemie 3 Personen infiziert sind. Bei Durchläufen mit nur einer inifizierten Person wurden manchmal keine anderen Personen rechtzeitig angesteckt, um die anfangs exponentielle Ausbreitung des Virus zu beginnen.

Für die mittlere Inkubationszeit α und Kontagiosität (Dauer der Ansteckungsfähigkeit) β haben wir die Werte $\alpha = 500$ Zeitschritte und $\beta = 1000$ Zeitschritte angenommen. Die Zeitschritte aus der Simulation können nicht direkt auf eine Zeiteinheit in der Realität übertragen werden, die Abläufe haben aber die gleichen relativen temporalen Verhältnisse. Deswegen ist es wichtig, dass das Verhältnis von Inkubationszeit und Kontagiosität mit echten Daten übereinstimmt. Wir haben als Referenz die Werte $\alpha = 5$ Tage und $\beta = 10$ Tage

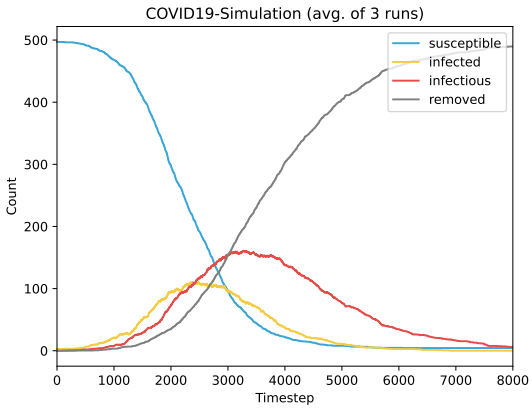


Figure 1. Status Counts (avg. über 3 Durchläufe). Unsere SEIR Ergebnisse stimmen mit typischen Ergebnissen des SIR-Modells (mit gewöhnlichen Differentialgleichungen) überein [8]

vom RKI (November 2021) verwendet [7] und unsere Parameter dementsprechend in einem Verhältnis $\frac{\alpha}{\beta} = 0.5$ gesetzt.

Das Projekt verwendet die Open-Source Python Projekte PyGame [21] (Spielengine) und PyMunk [22] (Physiksimulation) als Grundlage für unseren Simulator.

IV Lösungsstrategien

1 Klassisches SEIR-Modell mit Differentialgleichungen

Die Population wird in die folgenden drei Zustände aufgeteilt:

- S = Susceptible *(gesund)*
- I = Infected *(infiziert und ansteckend)*
- R = Recovered/ Removed *(nicht mehr ansteckend)*

Die Änderungsraten für die Anzahl der Partikel in jedem Zustand können wir in einem Differentialgleichungssystem für das SRI-Modell wie folgt formulieren [9] [10]:

$$\frac{d}{dt}S = -\beta\frac{SI}{N} \tag{1}$$

$$\frac{d}{dt}I = \beta\frac{SI}{N} - \gamma I \tag{2}$$

$$\frac{d}{dt}R = \gamma I \tag{3}$$

Hierbei wird keine Unterscheidung in die Klassen Exposed und Infectious benötigt, da die Unterteilung schon implizit mit den Änderungsraten modelliert wird (wenn mehr Personen infektiös sind, gibt es mehr Statusübergänge von Susceptible nach Infected).

Die Differentialgleichungen haben einen sehr limitierten Einblick in die simulierte Welt und Population (sie dienen nur als generische Beschreibung der Zustandsänderungen), deswegen werden wir uns nicht genauer mit dem SRI-Modell in der Form von Differentialgleichungen beschäftigen und gehen zu dem SEIR-Modell [11] als Partikelsimulation über.

2 SEIR-Modell mit Partikeln

Ein Partikel ist zu jedem Zeitpunkt in genau einem der folgenden Zustände:

- S = Susceptible *(gesund)*
- E = Exposed *(infiziert aber nicht ansteckend)*
- I = Infected *(infiziert und ansteckend)*
- R = Recovered/ Removed *(nicht mehr ansteckend)*

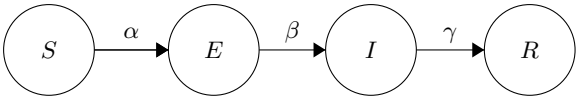
Wir definieren die Mengen S_t, E_t, I_t, R_t als die Mengen der Partikel, die sich zu einem Zeitschritt t im jeweiligen Zustand befinden. Da das SEIR-Modell annimmt, dass die Population eine feste Größe N hat, gilt für alle Zeitschritte:

$$N = |S_t| + |E_t| + |I_t| + |R_t| \tag{4}$$

Im klassischen SEIR-Modell wird die Anzahl der Zustandswechsel der Partikel mit Raten ausgerechnet [9]. Wir benutzen für unsere Partikelsimulation stattdessen Wahrscheinlichkeiten, die ausdrücken, wie wahrscheinlich es ist, dass ein Partikel in einem Zeitschritt t seinen Zustand wechselt:

- α = Infektionswahrscheinlichkeit
- β = Inkubationswahrscheinlichkeit
- γ = Erholungswahrscheinlichkeit

Die Wahrscheinlichkeit α, dass sich ein Partikel ansteckt, wird bei uns anders als beim konventionellen SIR-Modell gehandhabt. Dort wird der Übergang vom Zustand S in den Zustand E als Rate festgelegt; wir definieren hingegen nur die Wahrscheinlichkeit, dass sich ein Partikel ansteckt, wenn es mit einem ansteckenden Partikel kollidiert. Das ist für eine Partikelsimulation notwendig und hat den Vorteil, dass Messungen in der echten Welt über die Ansteckungswahrscheinlichkeit (beispielsweise mit oder ohne Maske) direkt als Parameter in die Simulation eingehen können, während beim konventionellen SEIR-Modell erst die allgemeine Rate modelliert werden muss, mit der die gesunde Population infiziert wird. Da wir die Partikel als Kreise mit einer Position und einem Radius modellieren, kann man auch den Ansteckungsradius über den Radius der Partikel im Modell anpassen.



Der Status wird für jedes infizierte Partikel (in einem der Zustände E oder I) einmal pro Zeitschritt zufällig mit den Wahrscheinlichkeiten β und γ aktualisiert. Ein Partikel im Zustand Exposed wechselt mit der Inkubationswahrscheinlichkeit β in den Zustand Infectious. Vom Zustand Infectious wechselt es mit der Erholungswahrscheinlichkeit γ in den Zustand Recovered, in welchem es dann für den Rest des simulierten Durchlaufs bleibt.

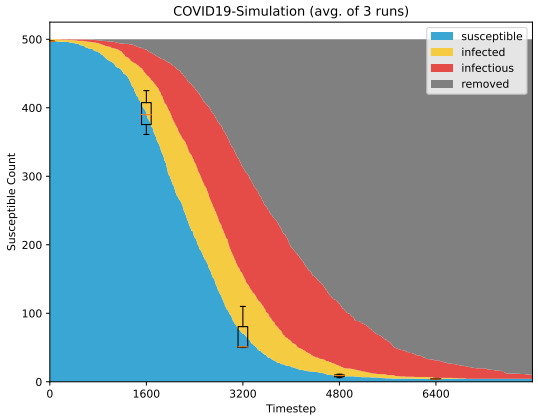


Figure 2. Status Stackplot mit Varianzbalken (avg. über 3 Durchläufe)

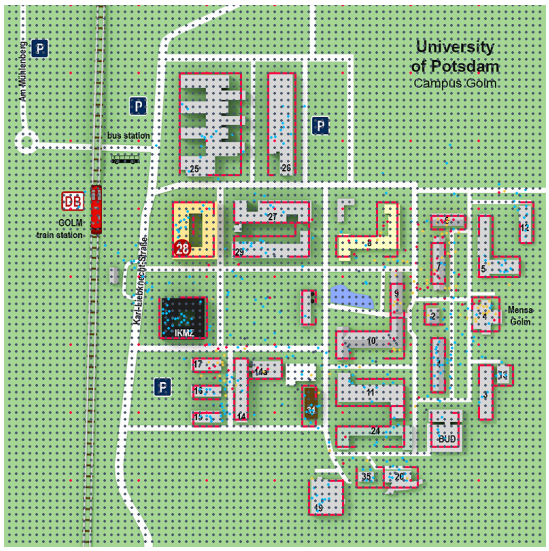


Figure 3. Simulator im Debug-Modus

V Aufbau der Simulation

1 Karte

Als Hintergrund der Simulation wird eine Karte von Golm genommen, die von der Webseite der Experimentalphysik stammt [12]. Für jedes Gebäude wurde ein Ursprungspunkt (Origin) festgelegt und Mauern gezogen, die als statische Objekte im PyMunk Simulator verwendet werden. Die Koordinaten der Mauern müssen alle einzeln in Relation zum Ursprungspunkt des Gebäudes gesetzt werden. Um die Koordinaten besser ermitteln zu können, haben wir einen Debug-Modus in den Simulator eingebaut, der alle 10 Zellen einen grauen Punkt und alle 100 Zellen einen roten Punkt zur Orientierung über die Karte legt. Der Modus kann in der Konfiguration des Simulators via '"debug_mode": True' aktiviert werden und wird in Figure 3 gezeigt.

2 Zug

Der Zug fährt periodisch auf der Karte von Norden nach Süden und wird dann wieder nach oben zurückgesetzt. Jeder Zyklus dauert 36.000 Pygame-Zeitschritte und wird durch eine Modulo-Rechnung mit dem aktuellen Zeitschritt der Simulation berechnet. Ein Zyklus enthält die folgenden Ereignisse, die relativ zum Beginn des Zyklus angegeben sind: Zu Beginn des Zyklus fährt der Zug mit einer Geschwindigkeit von -1,1 Pixel pro Sekunde in x-Richtung (leicht nach links) und 30 Pixel pro Sekunde in y-Richtung (nach unten). Bei 9.000 Zeitschritten hält der Zug an der Bahnstation an, öffnet die Tür und bleibt stehen. Bei 13.000 Zeitschritten schließt der Zug die Tür und fährt mit der ursprünglichen Geschwindigkeit weiter. Bei 36.000 Zeitschritten kehrt der Zug zum Startpunkt oben auf der Karte zurück und startet den nächsten Zyklus.

Es kann weiterführend modellieren werden, dass tatsächlich Personen mit dem Zug fahren. Allerdings haben wir für unser Projekt hier eine Grenze für den Entwicklungsaufwand gezogen und dieses Feature nicht implementiert.

VI Kollisionsbehandlung und probabilistische Infektionsübertragung

In PyMunk werden Kollisionen durch mehrere Funktionen behandelt, wobei die erste der Funktionen "collision_begin" ist. Die anderen Funktionen kümmern sich um die Berechnung der neuen Geschwindigkeiten der beiden Körper und Eigenschaften wie Reibung, Schaden an dem Körper usw. Diese Funktionalitäten sind der Hauptgrund, warum wir die Partikelsimulation nicht grundlegend neu (*"from scratch"*) geschrieben haben. In der PyMunk Bibliothek werden sehr effiziente Algorithmen verwendet, die durch die Programmiersprace C effizient sind und dadurch auch große Mengen von Partikeln handhaben können. Um den Simulator für unser Projekt benutzen zu können, haben wir eine eigene Implementierung der Funktion "collision_begin" geschrieben, die die Infektionsübertragung zwischen Personen simuliert. Die Funktion überprüft zunächst, ob eine der beteiligten Person an der Kollision als infektiös markiert ist (d.h. ihre Dichte ist 0.8). Wir mussten hier die Implementation etwas unüblich "hacken" und die Information über den Status einer Person in der Dichte des Partikels kodieren, da wir sonst nicht mit dem Simulator-Objekt und PyGame kommunizieren konnten. Falls eine der Personen infektiös ist, wird für jede andere beteiligte Person (deren Dichte 1.0 ist, was für einen gesunden Zustand steht) per Zufall mit der Infektionswahrscheinlichkeit α berechnet ob die gesunde Person infiziert wird oder nicht. Wenn die Person infiziert wird, wird ihre Dichte auf 0.9 gesetzt, um anzuzeigen, dass sie jetzt infiziert ist. Der tatsächliche Status der Infektion (als Attribut des Personen-Objekts) wird dann im nächsten Zeitschritt automatisch über die gegebene Dichte aktualisiert.

In unserer Implementierung wird zusätzlich der Ort der Infektionsübertragung gespeichert, indem die Position des infizierten Körpers in eine Liste "collision_points" eingefügt wird. Diese Information kann später verwendet werden, um eine Heatmap aller Übertragungsorte zu erstellen. Dort kann man dann ablesen, welche Orte auf der Karte besonders häufig zu Ansteckungen führen.

Die Klasse "Person" hat die Methode "update_infection_status", die für jede Person in jedem Zeitpunkt aufgerufen wird und die zufällig den Status der Person gemäß dem SEIR-Modell mit den Wahrscheinlichkeiten

β und γ aktualisiert (siehe "SEIR-Modell mit Partikeln"). Dabei werden die durchschnittlichen Zeiten für Inkubation und Ansteckung sowie der aktuelle Zeitpunkt übergeben. Diese Methoden ermöglichen es uns, eine probabilistische Veränderung des Infektionsstatus einer Personen im Laufe der Zeit zu simulieren.

1 Probabilistische Zielauswahl

Die Agenten besuchen alle Gebäude außer der Mensa und Bibliothek (diese beiden Ziele werden leicht bevorzugt) mit einer gleichverteilten Wahrscheinlichkeit. Die Wahrscheinlichkeit, dass eine Person eines der 30 Gebäude als nächstes Ziel X auswählt, wird wie folgt berechnet:

$$P(X = x_i) = \begin{cases} \frac{1}{10} & \text{falls } i \in \{3, 27\} \text{ (Mensa oder Bibliothek)} \\ \frac{1}{35} & \text{ansonsten} \end{cases} \tag{5}$$

Die Wahrscheinlichkeit P ist normiert, da

$$2 \times \frac{1}{10} + (30 - 2) \times \frac{1}{35} = 1. \tag{6}$$

Ein Agent verfolgt sein aktuelles Ziel für eine Anzahl von Zeitschritten, die auch normalverteilt bestimmt wird und sich für jedes neue Ziel ändert.

2 Goal-Based Vector Flow Field Pathfinding

Wir diskretisieren die Karte von Golm in ein 2D-Array mit der Form 800×800. Jeder Agent bewegt sich mit kontinuierlichen Koordinaten. Für das Nachschauen des optimalen Richtungsvektors zum Ziel wird eine diskrete Position verwendet, die wir durch einfaches Runden der Koordinaten auf ganze Zahlen erreichen. Die Zellen des Arrays definieren die Bereiche, in denen sich Agenten bewegen können, dabei kann eine Zelle entweder für Agents passierbar oder Teil einer Wand oder des Zuges sein (in dem Fall ist sie nicht passierbar).

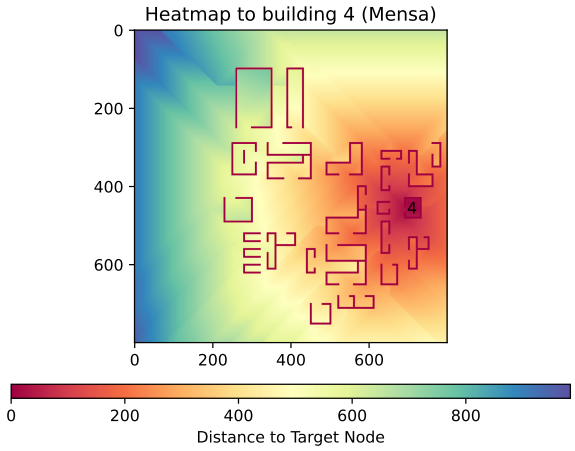


Figure 4. Visualisierung der Heatmap zur Mensa (speichert für jede Zelle die kürzeste Distanz zum Zielgebäude)

Agents bewegen sich durch die Map, indem sie die Distanz zu ihrem Ziel in einer Heat-Map nachschlagen, welche die Distanzen von jeder Zelle zu einem Target (Zielgebäude) speichert (Für eine Illustration der Heat-Map siehe Figure 4).

Die Heat-Map wird nur einmal berechnet und in allen folgenden Simulationen einfach als numpy-file geladen. Dort speichern wir das Ergebnis in einem 3D-Tensor mit der Form $30 \times 800 \times 800$, da pro Ziel eine 2D Heat-Map erstellt wird und wir diese Heat-Maps stacken. Das Stacken geschieht in der Implementation in der 0. Dimension, deswegen kann im Heat-Map Tensor über den Index der Dimension 0 das Gebäude bzw. Ziel adressiert werden.

Während der Ausführung sehr schnell der nächste Schritt des Flow-Paths (das ist der Weg zum Ziel) berechnet werden, denn die Distanzen der Nachbarn vom aktuellen Ort zum Ziel müssen nur miteinander verglichen werden, berechnet wurden sie schon. Das bedeutet, dass das Pathfinding zur Ausführung der Simulation keine aufwändige Rechenzeit verlangt und Agenten direkt den optimalen Weg von ihrer Position zu dem aktuellen Ziel bestimmen können.

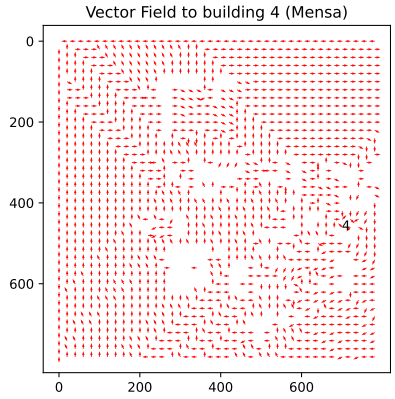


Figure 5. Vector Flow-Field zur Mensa. *Alle 10 Zellen wird illustrativ ein Vektor gezeigt, das tatsächliche Vektorfeld hat eine Größe von* 800×800

Ein Flow Field ist eine 2D-Matrix, die für jede Zelle des Grids einen Vektor enthält (Für eine Illustration siehe Figure 5). Dieser Vektor zeigt die optimale Richtung, um von dieser Zelle aus ein gegebenes Ziel zu erreichen. Flow Fields zusätzlich zu berechnen hat den Vorteil, dass während der Simulation noch weniger Rechenzeit benötigt wird, aber den Nachteil, dass sie entweder zusätzlich zur Heat-Map abgespeichert werden müssen (was viel Speicherplatz benötigt) oder die Heat-Map nicht direkt für Plotting-Zwecke rekonstruiert werden kann, sondern erneut berechnet werden muss. Deswegen berechnen wir im Gegensatz zu allen aufgelisteten Quellen zum Flow-Field Pathfinding das Flow-Field dynamisch wie beschrieben aus den Heat-Map der Distanzen, was aber eher ein kleines Implementationsdetail ist und fast keine Rechenzeit beansprucht.

Ein Punkt, der für beispielsweise Weiterentwicklungen der Simulation eine Rolle spielt ist, dass man die Heat-Map aktualisieren (neu berechnen) müsste, wenn sich die begehbaren Bereiche des Grids oder die Zielpunkten ändern - was bei uns nicht mehr der Fall sein sollte, da wir bereits alle Gebäude auf

der Karte eingebaut haben. Es gibt noch die Möglichkeit, die Gehwege abzugrenzen, aber unserer Erfahrung nach ist es realistischer, wenn die Studenten auch über den Rasen gehen und alle möglichen Abkürzungen nehmen können.

Der Flow-Field Algorithmus basiert auf dem Konzept des "potential fields", bei dem jede Zelle des Grids ein potenzielles Ziel hat. Dieses Ziel ist entweder explizit ausgezeichnet (z.B. ein bestimmtes Gebäude), oder kann auch einfach ein beliebiger Punkt auf der Karte sein (z.B. der Mittelpunkt des Campus). Der Algorithmus berechnet die optimale Pfadrichtung in jeder Zelle, indem er die Richtung zum Ziel mit der größten Abnahme des "Potentials" (der Distanz) berechnet.

Der Algorithmus beginnt mit einer Initialisierung der Potentiale in jeder Zelle des Grids. Dies kann entweder durch eine Distanzberechnung von jeder Zelle zum Ziel oder durch eine Verwendung von Vorwärts- und Rückwärts-Suchalgorithmen erfolgen. Für die Berrechnung der Potentiale wird die *Manhattan-Distanz* oder die *Euklidische Distanz* benutzt. In unserer Simulation haben wir die Euklidische Distanz berechnet, die in unserem Fall kürzere Wege als die Manhattan-Distanz liefert, da die Partikel kontinuierliche Koordinaten benutzen und dadurch nicht auf die diskreten {Hoch, Runter, Links, Rechts} Züge beschränkt sind, sondern sich in alle möglichen Richtungen von der aktuellen Position aus bewegen können.

Die Manhattan- und die Euklidische Distanz zwischen zwei Punkten p und q sind auf einem Gitter wie folgt definiert:

Manhattan-Distanz:

$$d_{\mathrm{m}}(p,q) = \sum_{i=1}^{n} |p_i - q_i| \tag{7}$$

Euklidische Distanz:

$$d_{\mathrm{e}}(p,q) = \sqrt{\sum_{i=1}^{n} (p_i - q_i)^2} \tag{8}$$

Anschließend wird das Potential in jeder Zelle durch eine Iteration des Algorithmus aktualisiert, bei der die Richtung zur Zelle mit dem niedrigsten Potential berechnet wird. Dieser Prozess wird solange wiederholt, bis sich das Potential in allen Zellen stabilisiert hat und ein statisches Flow Field erstellt wurde. Für unsere Simulation reicht es vollkommen aus, wenn wir ein partielles Flow-Field während der Laufzeit der Simulation berrechnen, da der Rechenaufwand eines partiellen Flow Fields für unsere relativ wenigen Partikel klein ist. Die optimale Richtung für eine derzeitige Position eines Partikels wird standardmäßig durch eine Kernel Convolution berechnet (bei unserer Implementation berechnen wir die Differenz der Koordinaten von der aktuellen Zelle und der Nachbarzelle mit der kleinsten Distanz zum Ziel - das ist äquivalent zu einer Convolution auf der Heat-Map). Das Partikel bewegt sich dann in die optimale Richtung, anschließend wird die Kernel Convolution wieder angewendet. So wird nur der Flow entlang des Weges errechnet und kein Flow Field für das ganze Feld. Eine Kernel Convolution ist eine Technik, bei der eine kleine Matrix (genannt Kernel) über eine größere Matrix (z.B. das Grid) verschoben wird. Jede Überlappung des Kernels mit der großen Matrix wird dann mit einer Kernel-Funktion verarbeitet, um ein Ergebnis zu berechnen. Im Fall des Flow-Field Algorithmus wird ein Kernel verwendet, um die optimale Richtung für jede Zelle zu bestimmen. Für die Simulation würde man einen kleinen $3x3$ Kernel benutzen, um die Distanzen der anliegenden Nachbarn zu vergleichen. Die Distanzen lassen sich aus unserem vorher berechneten Tensor ablesen.

Ein großer Vorteil des Flow Field Algorithmus ist, dass er auf Probleme mit vielen tausend Partikeln skalierbar ist. Er kann sowohl für kleine, einfache Karten als auch für große, komplexe Karten verwendet werden, und die optimale Richtung kann auch problemlos für viele Ziele gleichzeitig berechnet werden. Dies ermöglicht es uns, in unserem Simulator von Golm nicht nur ein statisches Flow Field für ein einzelnes Ziel zu berechnen, sondern auch für jedes Gebäude auf dem Campus, so dass die Agenten flexibel zwischen Zielen wechseln können. Ein weiterer Vorteil ist, dass der Flow Field Algorithmus in der Lage ist, Hindernisse und Barrieren in die Pfadfindung einzubeziehen. Indem wir die Zellen des Grids, die durch Wände oder andere Hindernisse blockiert sind, als unbegehbar markieren, kann der Algorithmus sicherstellen, dass Agenten keine unmöglichen Wege auf ihrem Weg zum Ziel nehmen. Hier kann es allerdings passieren, dass Agenten in Ausnahmesituationen stecken bleiben. Ein Beispiel wäre eine Ecke, bei der der diagonale Weg am kürzesten ist, aber eine Zelle anliegend an der Diagonale von einer Wand blockiert ist. Da der Agent dem kürzesten Weg folgen möchte, kann es passieren, dass er mit der Wand kollidiert. Um diese Situation größtenteils zu vermeiden, haben wir auch die direkt anligenden Zellen einer Wand für das Pathfinding als nicht passierbar markiert (eine Wand erscheint im Pathfinding also um einen Pixel breiter als sie tatsächlich im Simulator ist).

Ein möglicher Nachteil des Flow Field Algorithmus ist, dass er sehr speicherintensiv sein kann. Da für jedes Ziel eine separate Matrix berechnet werden muss, kann der Speicherbedarf schnell ansteigen, insbesondere bei großen Karten und vielen Zielen. Auch die Initialisierung des Potentials und die Iteration des Algorithmus kann einige Zeit in Anspruch nehmen, insbesondere bei großen und komplexen Karten. Bei uns beträgt die Rechenzeit für die Heat-Map wenige Minuten und der Tensor ist circa 150 MB groß.

Insgesamt ist der Flow-Field Algorithmus eine sehr effektive und skalierbare Methode als Lösung für ein Pathfinding-Problem. Er ermöglicht uns, in unserem Simulator ein realistisches und effizientes Bewegungsverhalten der Agenten zu simulieren, während er gleichzeitig die Rechenzeit während der Simulation minimiert und für kleine als auch für große Karten und einige Ziele gleichzeitig verwendet werden kann, wenn Agenten miteinander kollidieren dürfen (was der Sinn unserer Simulation ist). Ist das nicht gegeben, muss man auf Lösungen für das Multi-Agent Pathfinding Problem [17] zurückgreifen, die deutlich schlechter skalieren und häufig nur für maximal wenige hundert Agents funktionieren. Hier müsste man auf state-of-the-art Algorithmen, wie Conflict-based Search (CBS) [18], Scheduling-based Search [19] oder Compilation/Reduction-based Search [20] verwenden, welche sich aufgrund der Skalierungsprobleme nicht für eine Partikelsimulation eignen und kompliziert für dynamische Zielauswahl

(neues Ziel auswählen wenn das vorherige Ziel erreicht wurde) zu implementieren sind. Der Flow-Field Algorithmus kann nach unseren explorativen Tests verhältnismäßig gut mit bis zu über 10.000 Partikeln umgehen (obwohl dann der Platz auf der Karte nicht mehr für alle Partikel ausreicht).

VII Codebasis

1 Libraries

Q: Welche Libraries haben wir benutzt und wie können sie installiert werden?

Unser Code verwendet folgende Bibliotheken:

1. scikit-image (für skimage.measure, benötigt für ein optionales Max-Pooling in der Pfadfindung, um die Auflösung der Map zu verringern und dadurch eine kleinere Heat-Map zu speichern)
2. pymunk (enthält grundlegende Funktionalitäten wie Kollisionsbehandlung für unsere Teilchensimulation)
3. numpy (für Arrays und nützliche Funktionen)
4. pygame (für die visuelle Darstellung der Simulation)
5. matplotlib (für das Plotting)
6. typing (für Typehints in den Funktionssignaturen)
7. seaborn (für Plotting)

Die angegebenen Bibliotheken können alle manuell über pip installiert werden, oder **im Demo-Notebook einfach mit der ersten Codezelle installiert werden** (dafür in der Zelle "install_dependencies = True" setzen).

2 Strukturierung und Aufgaben der Python-Dateien

A Demo-Notebook

Die einzige Benutzerschnittstelle ist das "demo.ipynb" Notebook, in dem die Konfiguration (das "config" dictionary) des Simulators eingestellt und anschließend der Simulator ausgeführt wird. Es werden für jeden Durchlauf automatisch drei plots erstellt:

1. Graphen für die Anzahl der Personen pro Klasse. Für ein Beispiel des Plots siehe Figure 1.

2. Ein Stackplot mit den Anzahlen der Personen pro Klasse. Dieses zeigt die gleichen Daten wie das 1. Plot, aber die Verteilung der Klassen ist leichter abzulesen. Falls mehrere Durchläufe ausgeführt wurden (n_runs > 1), legt unser Code zusätzlich ein Boxplot über die "Susceptible" Klasse im Stackplot, an denen die folgenden zusätzlichen Informationen abgelesen werden können: Der Median der Durchläufe wird durch eine orangene Linie gekennzeichnet, die ersten (25%) und dritten (75%) Quartile werden durch die Enden der Box angezeigt und die minimalen und maximalen Datenpunkte werden durch die beiden Enden der äußeren Linien (über und unter der Box) dargestellt. Anhand dieser Daten kann man leicht erkennen, ob die einzelnen Durchläufe grob die gleichen Ergebnisse liefern oder ob sie stark voneinander abweichen. Wir zeigen diese Informationen nur für eine der Klassen, um die Lesbarkeit des Plots zu gewährleisten. Die Magnituden der Abweichungen für die anderen drei Klassen lässt sich grob anhand der Boxen dieser einen Klasse ableiten. Wenn die Werte der "Susceptible" Klasse eine hohe Varianz haben, hat dementsprechend auch die Summe der Werte der anderen drei Klassen eine höhere Varianz, da die Populationsgröße gefixt ist (Siehe Formel 4). Ein Beispiel für dieses Plot ist Figure 2.

3. Für die Summe der Durchläufe wird eine Heatmap mit den Kollisionen, die zu einer Übertragung der Infektion führen, generiert. Für dieses Plot haben wir die Positionen aller infektiösen Kollisionen gespeichert und plotten diese mit einem Kernel Density Estimate (KDE) Plot, das die Verteilung infektiöser Kollisionen zeigt. Anhand dieser Verteilung kann abgelesen werden, welche Gebiete auf der Karte häufig zu Infektionsübertragungen führen. In unseren Experimenten waren das besonders der Wegabschnitt zwischen dem Mathematikgebäude und der Mensa und die Kreuzungen von 4 Wegen (vor der Bibliothek und vor dem Physikgebäude). Für ein Beispiel siehe Figure 6.

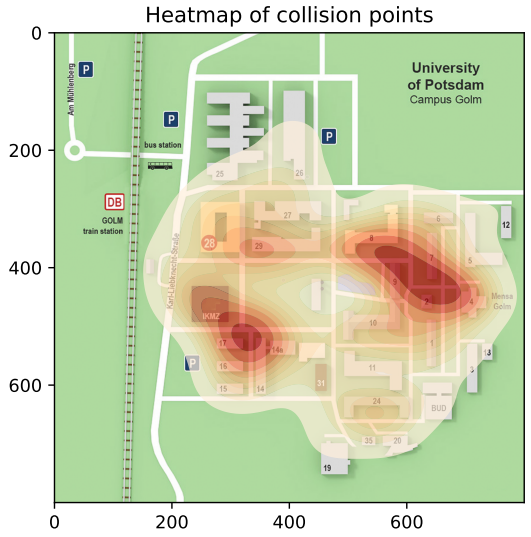


Figure 6. Heatmap für die infektiösen Kollisionen der Partikel: Besonders an den zentralen Wegkreuzungen, Engpässen und vor der Mensa treten vermehrt Infektionsübertragungen auf.

Logging der Experimente

Für jedes durchgeführte Experiment sollte der Benutzer in der Konfiguration einen Namen angeben. Es wird automatisch im "plots" Ordner ein Unterordner mit dem Namen des Experiments angelegt. Dieser beinhaltet die drei Plots und eine "conFiguretxt" Datei, in der die Ausführungsdauer und Konfiguration des Experiments abgespeichert werden. Diese Datei ist nützlich, wenn man später Ergebnisse reproduzieren möchte und die Parameter dafür benötigt. Aus diesem Grund setzen wir

für jedes Experiment auch explizit einen Seed, der die pseudo-zufällige Generierung der verwendeten Zufallsvariablen (z.B. die Startpunkte der Partikel) reproduzierbar macht.

B Objects.py

Die ”objects.py” Datei enthält Klassen für die Objekte, die in der Pymunk-Simulation verwendet werden. Das sind die Personen, Wände und der Zug.

C Pathfinding.py

In der ”pathfinding.py” Datei befinden sich Klassen, die für das Pathfinding benutzt werden. Einmal brauchen wir Nodes, die die Koordinaten und Distanz zum Zielpunkt von einer Zelle der Grid speichern, um einen Suchbaum aufzubauen. Der Suchbaum wird als Queue gehandhabt und die Pathfinder-Klasse enthält alle abstrakten Funktionen und Attribute für das Pathfinding. Dazu zählen die Zielpunkte der Gebäude, das Initialisieren des World Arrays und das Berechnen und Laden des Heat-Map Tensors.

D Simulator.py

Die ”simulator.py” Datei enthält die CovidSim Klasse, die alle anderen Aspekte (Pathfinding, Objekte erstellen, Partikelsimulation aufbauen, etc.) abstrakt handhabt. Der Simulator wird mit Parametern initialisiert und kann anschließend mit der ”run”-Funktion ausgeführt werden. Die Klasse beinhaltet Funktionen zum Erstellen des World-Arrays mit Grenzen und Objekten (besonders das Erstellen aller Mauern nimmt viele Zeilen Code in Anspruch), zur Kollisionsbehandlung (und Übertragung der Infektionen), zum Speichern von Informationen zum Plotting (Anzahl der Personen pro Status-Klasse), zum Updaten und der Visualisierung der Simulation für jeden Zeitschritt und zum generellen Aufbau eines Fensters. Dazu gehören das Erstellen des Fensters mit Icon, Tiel etc. und die Behandlung von Events (falls die Simulation beispielsweise über den ”X”-Button oder die ”ESC”-Taste geschlossen wird).

E Plotting-Notebook

Das ”plotting.ipynb” Notebook haben wir verwendet, um Plots zur Visualisierung des Flow-Field Pathfidings für diesen Report zu erstellen.

VIII Zusammenfassung

Unser Projekt simuliert einen Ausbruch der COVID-19-Pandemie auf dem Golmer Campus der Universität Potsdam. Dazu nutzen wir das SEIR-Modell und eine Partikelsimulation, unterstützt von einem Flow-Field Pathfinding Algorithmus, um die Wege für hunderte von Agenten schnell und dynamisch zu berechnen. Im Vergleich zum häufiger verwendeten SIR-Modell in Form von gewöhnlichen Differentialgleichungen, bietet das SEIR-Modell als Partikelsimulation eine detailliertere Möglichkeit für die Analyse der generierten Daten - beispielsweise der häufigsten Infektionsgebiete.

Diese Simulation kann als Notebook verwendet werden, um verschiedene Konfigurationen der Parameter zu vergleichen und automatisch Plots und Logs der Ausführungszeit und der Parameter zu erstellen. Das Projekt ist ein erster Schritt in die richtige Richtung, um informierte Entscheidungen bezüglich Eindämmungsmaßnahmen, Online-Lehre und Maskenpflicht zu treffen. Bevor es jedoch tatsächlich als Entscheidungshilfe verwendet wird, muss der Realitätsgrad durch Vergleich mit echten Daten der Pandemie ermittelt werden. Außerdem gibt es viel Potenzial zur Erweiterung durch die Integration von echten Daten, sowie die Anzahl der Personen pro Fakultät, demografischen Daten und Vorlesungszeiten.

Acknowledgment

Wir würden gerne Dr. Ralf Tönjes dafür danken, uns das Projekt in dem geplanten Umfang durchführen zu lassen.

References

[1] “Covid-19 vs. sars: How do they differ?” *Healthline*, 2022, visited on 29.01.2023. [Online]. Available: https://www.healthline.com/health/coronavirus-vs-sars

[2] “Was sind zoonosen?” *zoonosen.net*, 2019, visited on 29.01.2023. [Online]. Available: https://zoonosen.net/zoonosenforschung/was-sind-zoonosen

[3] D. Adam, “Special report: The simulations driving the world’s response to covid-19,” *Nature News Feature*, 2020, visited on 29.01.2023. [Online]. Available: https://www.nature.com/articles/d41586-020-01003-6

[4] P. G. e. a. Walker, “The global impact of covid-19 and strategies for mitigation and suppression,” *Imperial College COVID-19 Response Team*, 2020, visited on 29.01.2023. [Online]. Available: https://www.imperial.ac.uk/media/imperial-college/medicine/sph/ide/gida-fellowships/Imperial-College-COVID19-Global-Impact-26-03-2020.pdf

[5] “Coronavirus sars-cov-2: Erhöhte ansteckungsgefahr durch omikron,” *Bundeszentrale für gesundheitliche Aufklärung*, 2022, visited on 29.01.2023. [Online]. Available: https://www.infektionsschutz.de/coronavirus/basisinformationen/coronavirus-sars-cov-2-ansteckung-und-uebertragung

[6] Cochrane, “Ffp2-/n95-maske vs. chirurgische einmalmaske zur reduktion der Übertragung von coronaviren,” *Rapid-Reviews*, 2021, visited on 29.01.2023. [Online]. Available: https://ebminfo.at/ffp2_maske_vs_chirurgische_einmalmaske

[7] Robert-Koch-Institut, “Epidemiologischer steckbrief zu sars-cov-2 und covid-19,” *RKI: Inkubationszeit und Kontagiosität Werte*, 2021, visited on 29.01.2023. [Online]. Available: https://www.rki.de/DE/Content/InfAZ/N/Neuartiges_Coronavirus/Steckbrief.html;jsessionid=A8FF6D63EFF098C955023909CBB7DE95.internet091?nn=13490888#doc13776792bodyText10

[8] D. Smith and L. Moore, “The sir model for spread of disease - the differential equation model,” *Mathematical Association of America*, 2023. [Online]. Available: https://www.maa.org/press/periodicals/loci/joma/the-sir-model-for-spread-of-disease-the-differential-equation-model

[9] M. Petrica and I. Popescu, “A modified sird model for covid19 spread prediction using ensemble neural networks,” *arXiv preprint arXiv:2203.00407*, 2020,

visited on 29.01.2023. [Online]. Available: https://arxiv.org/pdf/2203.00407.pdf

[10] J. Daigle, "The sir model of epidemics," *Jay's Blog*, March 2020, visited on 29.01.2023. [Online]. Available: https://jaydaigle.net/blog/the-sir-model-of-epidemics/

[11] P. Heidrich, "Analysis and numerical simulations of epidemic models on the example of covid-19 and dengue," Doctor of Natural Sciences thesis, Universität Koblenz-Landau, 2021. [Online]. Available: https://kola.opus.hbz-nrw.de/frontdoor/deliver/index/docId/2185/file/DissertationHeidrich_pub.pdf

[12] "Golm map," 2023, visited on 10.02.2023. [Online]. Available: http://exph.physik.uni-potsdam.de/howto.html

[13] L. Erkenbrach, "Flow field pathfinding," *Leif Erkenbrach's programming blog*, 2013, visited on 29.01.2023. [Online]. Available: https://leifnode.com/2013/12/flow-field-pathfinding/

[14] S. Durant, "Understanding goal-based vector field pathfinding," *Game Development*, 2013, visited on 29.01.2023. [Online]. Available: https://gamedevelopment.tutsplus.com/tutorials/understanding-goal-based-vector-field-pathfinding--gamedev-9007

[15] E. Emerson, "Crowd pathfinding and steering using flow field tiles," *GameAIPro*, 2020, visited on 29.01.2023. [Online]. Available: http://www.gameaipro.com/GameAIPro/GameAIPro_Chapter23_Crowd_Pathfinding_and_Steering_Using_Flow_Field_Tiles.pdf

[16] G. Pentheny, "Efficient crowd simulation for mobile games," *GameAIPro*, 2020, visited on 29.01.2023. [Online]. Available: http://www.gameaipro.com/GameAIPro/GameAIPro_Chapter24_Efficient_Crowd_Simulation_for_Mobile_Games.pdf

[17] R. Stern, N. Sturtevant, A. Felner, S. Koenig, H. Ma, T. Walker, J. Li, D. Atzmon, L. Cohen, T. K. S. Kumar, E. Boyarski, and R. Bartak, "Multi-agent pathfinding: Definitions, variants, and benchmarks," in *AAAI/ACM Conference on AI, Mobility, and Autonomous Systems*, 2019, pp. 75–82. [Online]. Available: https://arxiv.org/abs/1906.08291

[18] G. Sharon, R. Stern, A. Felner, and N. R. Sturtevant, "Conflict-based search for optimal multi-agent pathfinding," *Artificial Intelligence*, vol. 219, pp. 40–66, 2015. [Online]. Available: https://www.aaai.org/ocs/index.php/AAAI/AAAI12/paper/viewFile/5062/5239

[19] J. Roman Barták and M. Vlk, "A scheduling-based approach to multi-agent path finding with weighted and capacitated arcs," 2018. [Online]. Available: https://ifaamas.org/Proceedings/aamas2018/pdfs/p748.pdf

[20] M. H. et. al., "Reduction-based solving of multi-agent pathfinding on large maps using graph pruning," in *AAMAS*, 2022. [Online]. Available: https://www.ifaamas.org/Proceedings/aamas2022/pdfs/p624.pdf

[21] P. Shinners, "Pygame documentation," https://www.pygame.org/docs/, 2011, visited on 10.02.2023.

[22] V. Blomqvist, "Pymunk," 2022, visited on 10.02.2023. [Online]. Available: http://www.pymunk.org/en/latest/

IX Appendix

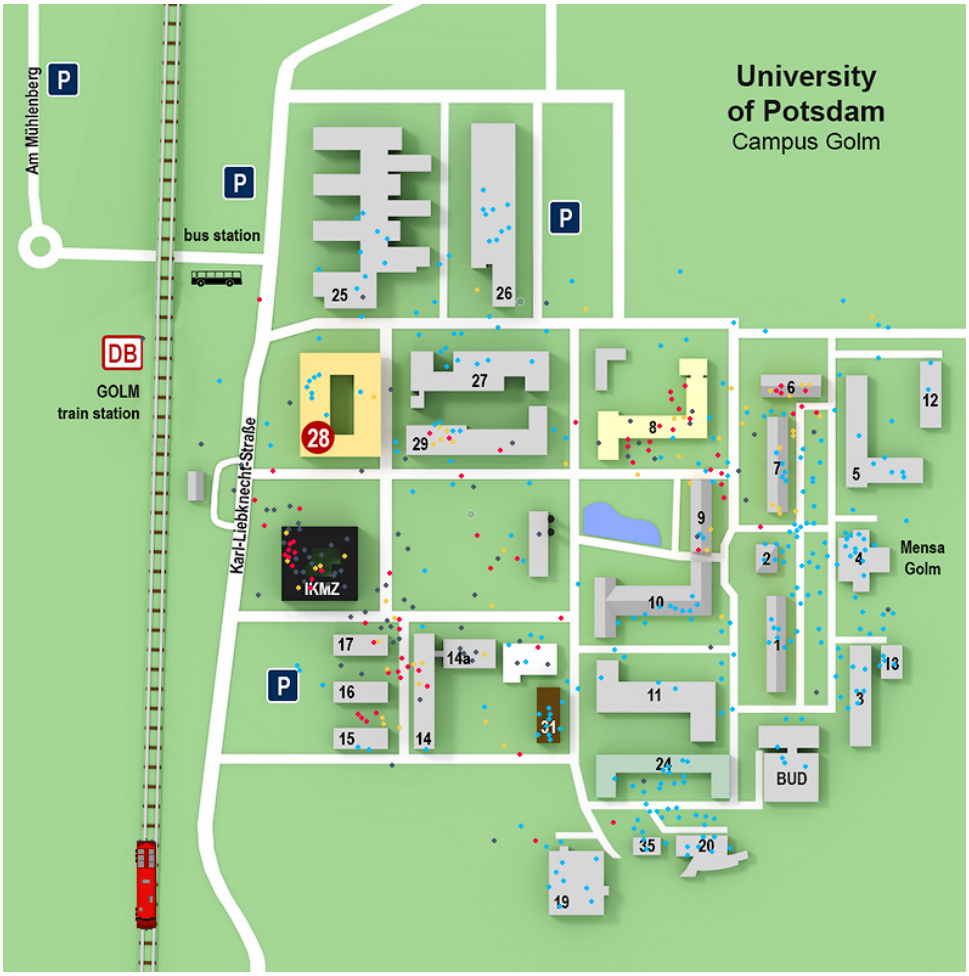


Figure 7. Screenshot des Simulators während eines Durchlaufs

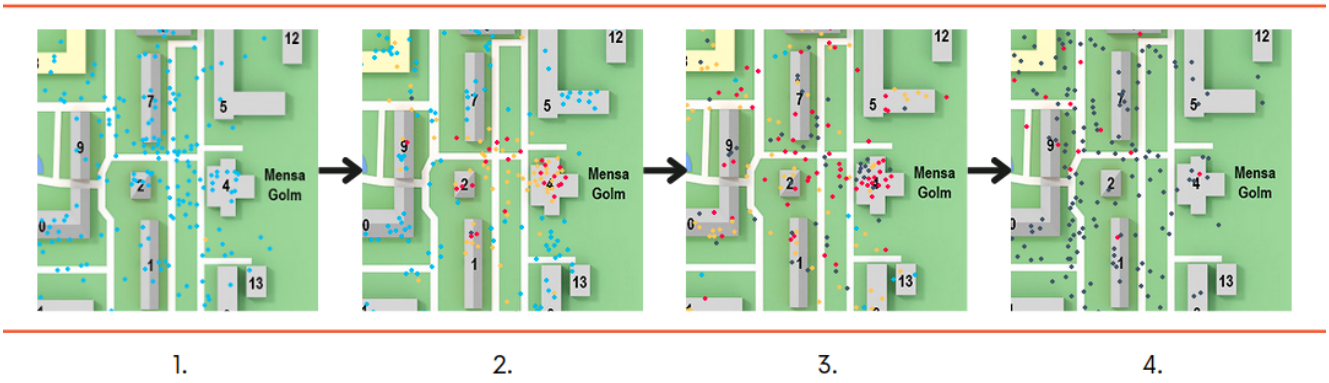


Figure 8. Illustratives Beispiel eines typischen Durchlaufs