Physik Snake Game

Table of Contents

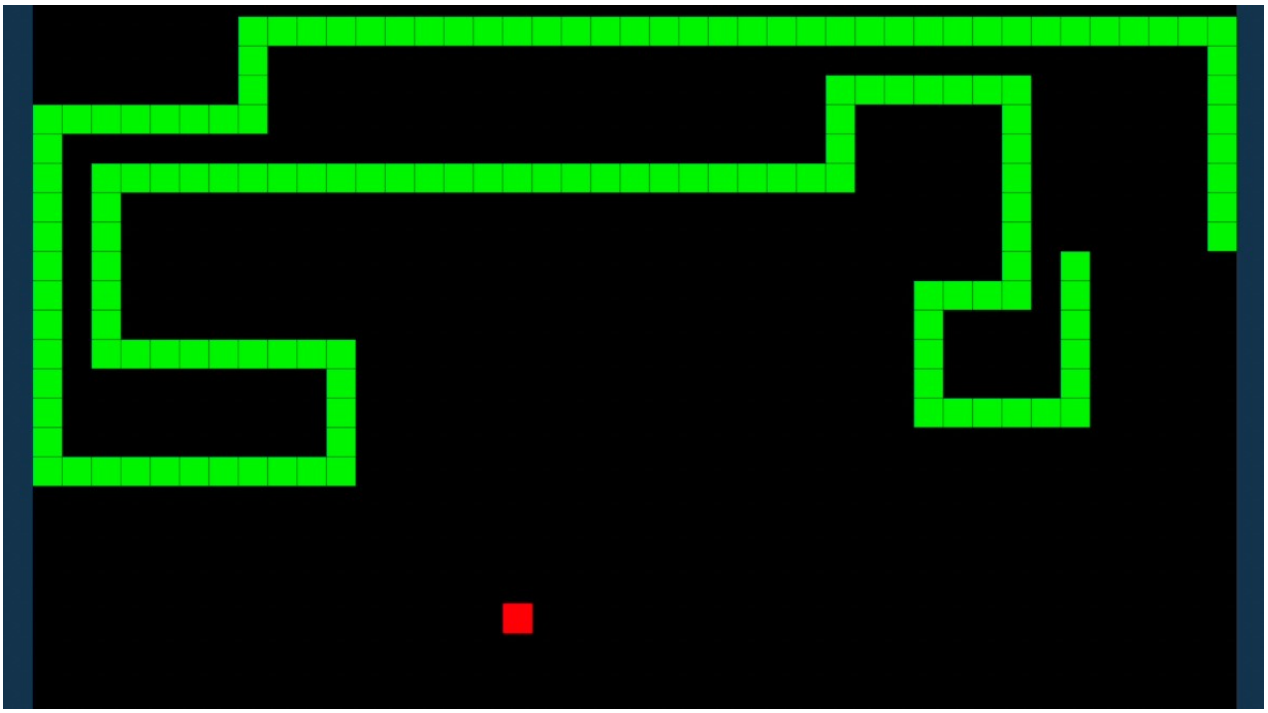
Einführung

Inspiration
https://www.youtube.com/watch?v=tjQIO1rqTBE
https://www.youtube.com/watch?v=TOpBcfbAgPg

Für eine originales Spielbare Version von Snake habe ich eine Pygame Implementation Übernommen. Der Autor ist Rajat Dipta Biswas https://github.com/rajatdiptabiswas/snake-pygame.
Ein eigenes Spiel komplett neu zu schreiben erscheint mir nicht interessant, daher nutze ich diese Basis.
Zielsetzung ist es diesen Code zu adaptieren und unterschiedliche Lösungsalgorithmen vorzustellen.

Was ist Snake

Snake ist ein Computerspielklassiker und kann grundsätzlich Geschicklichkeit Spiel gesehen werden, bei dem der Spieler (oder Akteur) eine namensgebende ‚Schlange‘ durch ein Spielfeld gesteuert werden kann. Unterschiedlich Adaptiert ist das Originale spiel einfach ein zweidimensionales Raster, wobei die Schlange anfangs ein einziges Feld einnimmt.
Auf den restlichen Feldern kann zufällig ein ‚Apfel‘ generiert werden. Der Apfel wird durch ein rotes Feld dargestellt. Sobald die Schlange den Apfel isst (berührt) verschwindet dieser und ein neuer Apfel wird generiert. Die Schlange wird mit dem Essen um genau ein Feld länger und

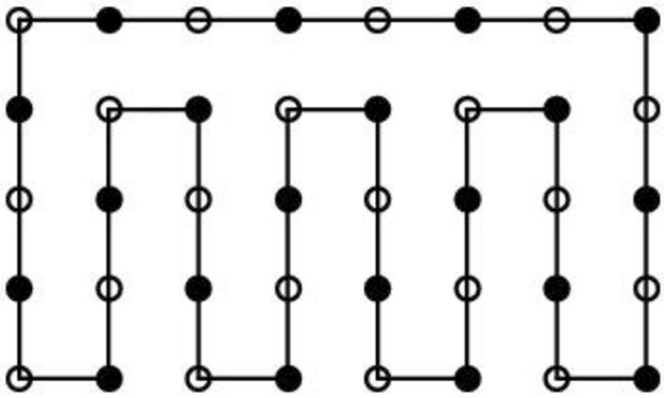


(image of Browsergame: https://www.coolmathgames.com/0-snake)

Snake als Graphentheorie

Ein Graph ist eine abstrakte Struktur G(V,E) welche Knoten V (vertices) und Kanten E (edges) definiert.

Pertubated Hamiltonian Cycle (John Tapsell)



https://www.researchgate.net/publication/51915862_Hamiltonian_Paths_in_Two_Classes_of_Grid_Graphs

Perfektes Spiel

Sicherheit

Im Rahmen dieser Arbeit spreche ich von einem Snake als Gewonnen, wenn die Schlange durch das Essen der Äpfel das Gesamte Spielfeld ausfüllt.
Ein Maßstab der Qualität eines Algorithmus, ist also wie viel Prozent des Spielfeldes ausgefüllt sind. Idealer weise werden hier immer 100% Erreicht.
Eine Möglichkeit dieses Ziel zu erreichen ist es ein Hamiltonkreis abzugehen (Algorithmus1). Jeder Hamiltonkreis garantiert 100% zu erreichen.

Schritte

Ein zweiter Maßstab an ein Algorithmus ist wie oft sich die Schlange einen Schritt weiter bewegt bevor das Spiel beendet ist.
Da die Position eines neuen Apfels auf einem Zufälligem freien Feld generiert wird, können unterschiedliche Algorithmen unterschiedlich schnell den Apfel erreichen. Wobei idealer weise sicher gestellt wird, das nach dem Maßstab der Sicherheit die Schlange in keine Sackgasse läuft.
Bei unserem Algorithmus 1, dem statischen Hamiltonkreis, wird zu beginn der Pfad bestimmt und nie neu angepasst. Daher muss die Schlange durchschnittlich die Hälfte der freien Felder abgehen, bevor sie den Apfel ist.
Für den ersten Apfel werden durchschnittlich 1+Felder/2 schritte benötigt für den zweiten Apfel 1+(Felder-1)/2 Schritte.
Die Erwartete Schrittzahl um das Spiel zu gewinnen sind daher

Sum von {n=0} bis {Felder-1} 1+(Felder-n)/2

Bei einer Spielfeldgröße von 72x48 gibt es 3456 Felder. Die Erwartete Schrittzahl bei einem einer einfachen Hamiltonkreisstrategie (siehe Kapitel Algorithmen) ist in diesem Fall 2990304 um mit Algorithmus 1 das Spiel zu gewinnen. Es werden durchschnittlich 865,25 Schritte benötigt um einen Apfel zu Essen. Ein Besser Algorithmus ist ein Algorithmus der eine kleinere Schrittzahl benötigt.

Aufgrund der enormen Schrittzahl muss jeder einzelner Schritt Effizient sein. Für das generieren eines Apfels benötigt man allerdings die Position des Körpers der Schlange, damit der Apfel nicht innerhalb generiert wird. Die vom Körper belegten Felder verändern sich mit jedem Schritt. Daher wird eine Liste der belegten Felder benötigt. 3 Millionen Operationen auf teilweise Langen Listen kann aber einige Zeit beanspruchen . Dieser Overhead verursacht 1-3 Sekunden über ein gesamtes Spiel.
Auffälliger noch ist die Brute-Force Variante die freien Felder zu berechnen, welche benötigt werden um einen neuen Apfel zu generieren.

```
free_spaces = [field for field in data["grid"] if field not in data["body"]]
```

Diese Funktion liegt in $O(n^3)$ da für jeden Apfel $O(n^1)$ eine Liste aus allen Feldern generiert wird $O(n^2)$ und für jedes Item die Schlange nach Überschneidungen durchsucht wird $O(n^3)$. In der Online übernommen Version von Rajat Dipta Biswas wurde daher akzeptiert, das der Apfel im Körper auftaucht.
Die Konvertierung des „bodys“ zu einem Set reduziert das Problem auf $O(n^2 \log n)$ und für den ersten einfachen Algorithmus ein bedeutet das ein speedup von ~250 Sekunden auf ~2,5 s

```
body = set(data["body"])
```

**Integration führt zu Speedup um Faktor ~100*

Diese Anpassung wirkt sich natürlich nicht auf die Lösungsalgorithmen selbst aus. Der Körper kann auch nicht permanent als Liste gespeichert werden, da die Rennfolge des Körpers wichtig ist.

Eine weitere Optimierung wäre es den Plan zu erstellen und die Felder dem Plan nach zu ordnen, damit könnte der Körper von z.B. Feld X - Feld Y definiert werden und nur diese zwei Zahlen mit

jedem Schritt verändert werden. Bei der Erstellung eines neuen Planes müsste die Ordnung angepasst werden. Wobei die Schwanzspitze der Schlange dann als erstes Feld der neuen Ordnung des Planes definiert werden müsste und der bisherige besetzte Weg als Reitfolge übernommen werden würde. #TODO Sinnvoll?

Zeit

Ein letzter Maßstab ist die Zeit die ein Algortmus benötigt um das Problem zu Lösen. Idealer weise kann das Spiel in Echtzeit Löungen finden. Da mit dem essen des Apfels erst der nächste Apfel generiert wird hat, hat ein Algorithmus unter Echtzeitbedingungen nur einen einzelnen Schritt Zeit um zu entscheiden welchen Pfad die Schlange einschlägt. Bei einer Geschwindigkeit von 50ms pro Schritt, bleiben also nur 50ms um ein NP-schweres Optimierungsprobelm zu Lösen.

Eine weitere Variante wäre es während des Rechnens einen Vorläufigen Pfad einzuschlagen und diesen anzuzupassen, wenn man einen schnelleren findet. Da sich die position des Kopfes aber alle 50ms ändert und damit der Algorthumus damit dynamisch umgehen muss, übersteigt diese Variante den Rahmen dieser Arbeit.

Um auch komplexere Algorithmen testen zu können wird beim Essen des Apfels das Spiel pausiert, bis der neue Pfad berechnet wurde. Die Zeit die Pausiert werden muss, ist der dritte Maßstab nach dem ein Algorithmus bewertet werden kann.

Schnittstelle/Backend

Darstellung

Eine überraschende Schwierigkeit birgt die Darstellung der Pläne. Insbesondere haben bestimmte implementieren maßgeblich Einfluss auf Komplexität und Effizienz. Wobei einige Algorithmen bestimmte Schnittstellen brauchen. Die je nach implementieren der Daten anders Umsetzbar sind. Auch wenn für einen einzelnen Algorithmus einfach Daten erhoben werden können, wird es komplexer wenn unterschiedliche Algorithmen vergleichbar bleiben sollen.
Auch die visuelle Darstellung stellt bestimmte Anforderungen die beachtet werden müssen.

Plan als Wegbeschreibung

Ein Ansatz der Darstellung eines berechneten Weges ist zu jedem Feld im Gitter eine Richtung anzugeben.

(0,1):hoch → (0,0):rechts → (1,0):rechts…

Vorteile:
- Kann einfach mit unvollständiger Wegen umgehen.
- Kann einzelne Schritte Überprüfen.
- Einfache dynamische Anpassung möglich
- Menschlicher Spielweise ähnlicher

Nachteile:
- Der Körper muss als Liste gespeichert werden (aufwändigerer Overhead).
- Apfel Generierung $O(n^2 \text{Log } n)$

Plan als Ordnung

Ein Weg kann auch ein geordnete Reihenfolge der Felder dargestellt werden. Hierfür können die Felder im Gitter in einer bestimmte Reihenfolge gespeichert werden, die dann im Lauf des Spiels abgegangen werden. Im falle eines statischen Hamiltonkreises können die Felder einmalig so geordnet werden, wie die Schlange den Fad ablaufen soll.

Feld 3455 = (0,1) → Feld 0 = (0,0) → Feld 1= (1,0) …

Bei dieser Variante kann die Schlange von Feld T (Tail) bis Feld H (Head) dargestellt werden. Also mit zwei Integer anstatt mit einer Liste. Dies ist aber nicht immer trivial. Denn sobald Schlange bei Feld 0 ankommt muss der Körper als zwei Listenteile gehandhabt werden (Tail bis Ende + Anfang bis Head). Diese Variante könnte zwar theoretisch einige Schritte beschleunigen, in der Praxis zeigt sich jedoch dies Option schwer Umsetzbar ist. Insbesondere wenn der Körper wiederholt als neue Liste erzeugt werden muss verliert dies Variante jede Sinnhaftigkeit.

Um diesen und weiteren Problemen aus dem Weg zu gehen bietet sich eine weniger intuitive Lösung an. Bei dieser Variante verschiebt sich nicht die Schlange sondern das Spielfeld. Wobei der Schwanz der Schlange immer bei Feld 0 liegt und die Länge N der Schlange die Position des Kopfes bei Schritt N-1 angibt.

Schritt 3455 = (0,1) → Schritt 0 = (0,0) → Schritt 1 = (1,0)…
Schritt 3455 = (0,0) → Schritt 0 = (1,0) → Schritt 1 = (2,0)…

Vorteile:
- Effiziente Darstellung der belegten und unbelegten Felder.
- Schritte mit Deque/Liste umsetzbar. #TODO vergleichen
- Leichter Zugriff und Anpassbarkeit freier Felder
- Apfel Generierung in O(n) möglich

Nachteile:
- Unituitiv

Der Umbau zu dieser Backend darstellung ermöglicht das komplette Lösen und ausführen von 3 Millionen Schritten mit dem einfachen statischen Hamiltonkreis in rund einer Sekunde. Gleichzeitig können Lösungsalgorithmen deutlich Dynamischer mit den Daten umgehen.

Visualisierung

Das vom Computer gespielte Snake Spiel kann ebenfalls visualisiert werden. Hierfür wurden teile des Codes von Rajat Dipta Biswas übernommen und angepasst um die Computer imputs zu verstehen.

Im Rahmen dieser Arbeit habe ich zwischen Benchmarks und Visualisierung getrennt, um Daten nicht zu verfälschen. Idealer weise wird ein Snake-Spiel zuerst gelöst die Daten erhoben und anschließend kann das Spiel jederzeit visuell aus den Daten rekonstruiert werden. Allerdings müssten dafür jeder Schritt oder jeder erstellte Plan gespeichert werden um sie dann visuell darstellen zu können. Um nicht unnötig viel Daten zu erheben werden entweder Benchmarks erstellt oder ein Spiel Visualisiert, nicht beides.

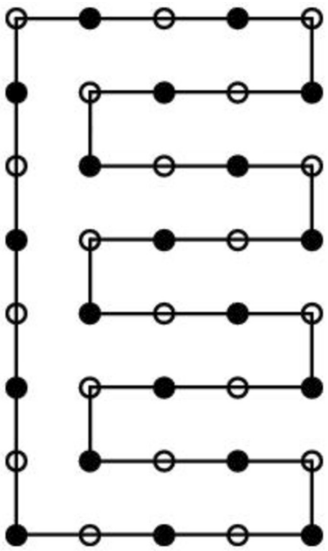
Algorithmen

Sobald die Schnittstelle des Spiel einen Pfad anfragt. Gibt ein Ausgwählter Algorthmus einen Pfad und die benötigte Rechenzeit zurück. Die Schnittstelle Testet dann ob der Algorithmus einen freien und regelkonformen Pfad angibt. In den Daten werden dann die Benchmark werte des algorithmus gespeichert und können verglichen werden.

Einfacher Hamiltonkreis

Nach dem ‚Grinberg's theorem‘ gelten In einem Gitter der Größe MxN folgende Aussagen:
1. Wenn M=1, N=1 dann gilt der Punkt im Gitter als Hamiltonkreis.
2. Wenn nicht (1) und M=1 xor N=1, gibt es kein Hamiltonkreis (es gibt keinen Weg zurück).

3. Wenn M und N ungerade ist der Graph nicht Hamiltonisch.
4. Wenn M oder N grade ist gibt es immer ein Hamiltonkreis.
Wobei wir uns für die einfachste Variante an folgendem Muster orientieren:



https://www.researchgate.net/publication/51915862_Hamiltonian_Paths_in_Two_Classes_of_Grid_Graphs

Statischer Hamiltonkreis

KI
Hamiltonian Cycle in ASP. https://www.cs.utexas.edu/users/vl/papers/wiasp.pdf