Physik Snake Game

Inhaltsverzeichnis

Einführung

Für dieses Projekt möchte ich ein spielbare Version von Snake bauen und vorstellen, die Schnitten für Algorithmen besitzt, die das Spiel so gut wie möglich Lösen und nach unterschiedlichen Maßstäben verglichen werden könne. Dabei soll es möglich sein den Lösungsalgorithmus beim spielen zuzusehen. Das Projekt kann auf Github eingesehen werden:
https://github.com/ZimneJonas/Snake-fastestPathNP

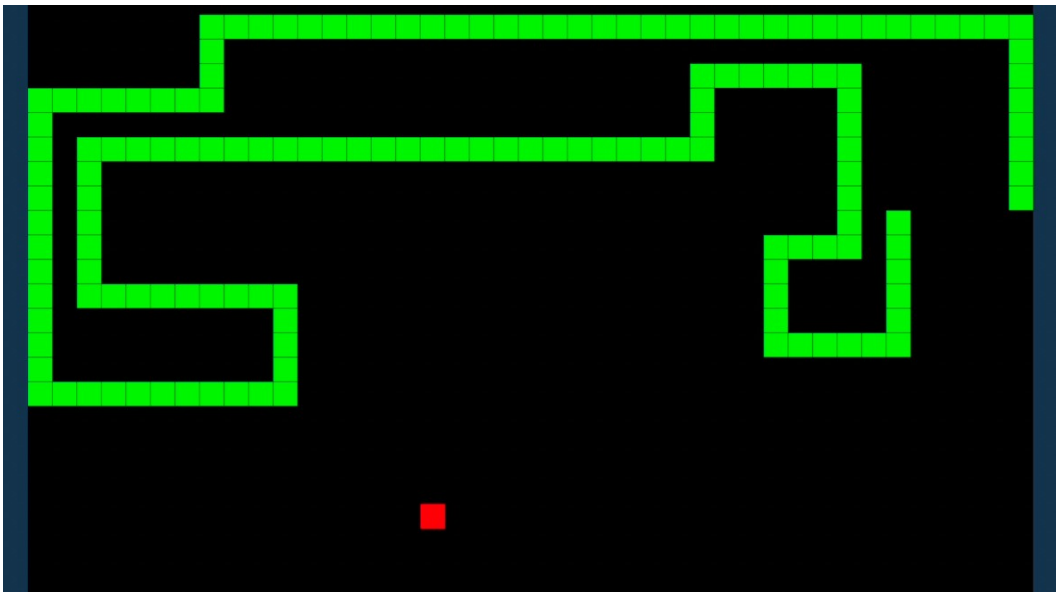
Für eine originales Spielbare Version von Snake habe ich eine Pygame Implementation Übernommen und angepasst. Der originale Autor ist Rajat Dipta Biswas. Ein eigenes Spiel komplett neu zu schreiben erscheint mir nicht interessant, daher wird der Code hier als Basis übernommen.
https://github.com/rajatdiptabiswas/snake-pygame

Zielsetzung ist es diesen Code zu adaptieren und unterschiedliche Lösungsalgorithmen vorzustellen. Insbesondere lag der Fokus dieser Arbeit darauf die Implementierung so zu schaffen, das sie flexibel und effizient ist, sodass ohne großen Aufwand neue Lösungsalgorithmen implementieren und getestet werden können. Wobei der Rechenaufwand des Spiels selbst so klein bleibt, dass Lösungsalgorithmen gut verglichen werden können.

Was ist Snake

Snake ist ein Computerspielklassiker und kann grundsätzlich als Geschicklichkeitsspiel gesehen werden, bei dem der Spieler (oder Akteur) eine namensgebende ‚Schlange‘ durch ein Spielfeld steuert. 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 berührt (isst) verschwindet dieser und ein neuer Apfel wird generiert. Die Schlange wird mit dem Essen des Apfels um genau ein Feld länger.



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

Snake als Graphentheorie

Das Snake-Spiel kann als Problem der Graphentheorie betrachtet werden, da es auf einem Gitternetz gespielt wird, das als Graph modelliert werden kann.

Die Knoten des Graphen stellen die Positionen auf dem Gitternetz dar, während die Kanten die möglichen Bewegungen des Schlangenkopfes von einer Position zur anderen darstellen. Ein einfaches 2D-Gitternetz kann als einfacher ungerichteter Graph dargestellt werden.

Die Bewegungen der Schlange können als Pfad auf dem Graphen modelliert werden. Wenn die Schlange sich bewegt, ändert sich der Pfad, indem ein Knoten hinzugefügt wird und der letzte Knoten entfernt wird, um die Länge der Schlange konstant zu halten. Der Pfad kann dann auf Äquivalenzklassen von zyklischen Permutationen reduziert werden, da der Startknoten des Pfades beliebig gewählt werden kann.

Mit Äquivalenzklassen von zyklischen Permutationen meint man die Menge aller Permutationen, die durch zyklische Verschiebungen einer gegebenen Permutation entstehen. In Bezug auf das Snake-Spiel bedeutet dies, dass der Pfad der Schlange auf dem Gitternetz durch eine Permutation der Knoten auf dem Pfad dargestellt werden kann.

Ein Beispiel: Angenommen, der Pfad der Schlange auf dem Gitternetz lautet [1, 2, 3, 4, 5]. Dann können die folgenden Permutationen des Pfads als äquivalent angesehen werden:

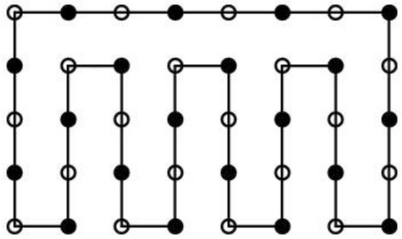
- [1, 2, 3, 4, 5]
- [2, 3, 4, 5, 1]
- [3, 4, 5, 1, 2]
- [4, 5, 1, 2, 3]
- [5, 1, 2, 3, 4]

Diese Permutationen unterscheiden sich in der Startposition der Schlange auf dem Pfad, aber sie repräsentieren alle den gleichen Pfad auf dem Gitternetz.

Um das Spiel zu lösen, kann man verschiedene Algorithmen der Graphentheorie verwenden. Zum Beispiel kann man einen Breitensuche-Algorithmus verwenden, um den kürzesten Pfad zu finden, den die Schlange nehmen kann, um das Futter zu erreichen. Man kann auch ein Hamilton-Kreis-Algorithmus verwenden, um zu überprüfen, ob es möglich ist, das Spiel zu gewinnen, indem man das gesamte Gitternetz abdeckt, ohne einen Knoten zweimal zu besuchen. Insbesondere darauf liegt der Fokus in dieser Arbeit.

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, z.B. wenn wir uns an folgendem Muster orientieren



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

$$\sum_{n=0}^{Felder-1} (1+(Felder-n)/2)$$

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 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 .

(Umsetzung im Kapitel: Backend/Schnittstellen)

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 voraussichtlich NP-schweres Optimierungsproblem zu Lösen (Hamiltonicity-Nachweis ist NP-schwer).

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.

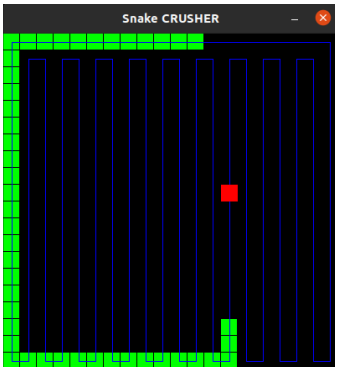
Um auch komplexere Algorithmen testen zu können wird beim Essen des Apfels das Spiel pausiert, bis der neue Pfad berechnet wurde. Die Zeit die Pausiert werden muss, ist der dritte Maßstab nach dem ein Algorithmus bewertet werden kann.

Schnittstellen/Backend

Um das Snake Spiel zu implementieren wurden Klassen verwendet. Um das Snake spiel Lösen startet man LogicSnake.py. In der Dort erstellten Oberklasse werden grundsätzliche Dinge angepasst, wie zum Beispiel die Größe des Spielfeldes. Insbesondere kann man dort auch entscheiden ob das Spiel während des Lösens gezeigt werden soll, oder ob nur die Zeiten erhoben werden sollen. Wenn das Spiel gezeigt wird, läuft es langsamer und ist durch die Angegebene FPS begrenzt. Die LogicSnake Klasse nutzt die Subklasse Info wo alle relevanten daten über das Spiel gespeichert werden. Info implementiert eine Datenbank sowie relevante Funktion für das laufende Spiel.
Ein Ziel war es diese Klassen flexibel genug zu halten um mit verschiedensten

Lösungsansetzen umgehen zu können und trotzdem keinen extremen Overhead zu verursachen, sodass mit tausenden Äpfeln und Millionen von Schritten schnell umgehangen werden kann und möglichst der Lösungsalgorithmus selbst der größte Limitierende Faktor ist. Die Info Klasse ist so angelegt das man eine grundsätzliche Ordnung angeben (blaue Linie) kann in der die Schlange das Feld abgehen soll. Mit einer Tour liste wird aber auch die Möglichkeit gelassen ein Pfad durch die Ordnung zu gehen die von der Ursprünglichen Ordnung abweicht.

Computertechnische Herausforderungen im Backend



Durch verschiedene Anpassungen konnte der Overhead auf Ca. 10^{-8} Sekunden pro Schritt reduziert werde und verursacht 1-3 Sekunden über ein gesamtes Spiel, bei einem relativ großem Spielfeld.

Rechenaufwand bei Apfelgenerierung

Die Bedeutung der Umsetzung ist spätestens dann relevant wenn man versucht mit einem Brute-Force Algorithmus 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 generiert werden kann. Für fehlerhafte Menschliche Spieler fällt dies selten auf, für Algorithmen, die zuverlässig das Spiel gewinnen ist ein solches Detail sehr relevant, da am Ende das Spielfeld immer mehr aus Schlange besteht.

Die Konvertierung des „bodys“ zu einem Set reduziert das Problem auf $O(n^2 \log n)$ und für den ersten einfachen Algorithmus bedeutet das ein speedup von 100 und damit von ~250 Sekunden auf ~2,5 s

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

Der Körper kann auch nicht permanent als Set gespeichert werden, da die Rennfolge des Körpers im Laufe des Spiels wichtig ist.

Rechenaufwand bei Schritten

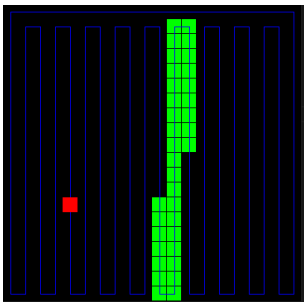
Für Effizienz sind auch einige Listen in der Datenbank als Warteschlange (Queue) implementiert. Da z.B. der Körper der Schlange immer vorne ein neues Feld eingefügt bekommt und hinten Felder verliert.
Eine Warteschlange ist eine Datenstruktur, die Elemente in einer bestimmten Reihenfolge speichert und diese Elemente nach dem FIFO-Prinzip (First-In-First-Out) verwaltet. Das bedeutet, dass das Element, das zuerst hinzugefügt wurde, auch zuerst entfernt wird. Solange dieses Prinzip eingehalten wird sind Operationen auf der Warteschlange in O(1). Während eine normale Listen in Python nicht so schnell löschen und anhängen können.

Derartige Verbesserungen beachten natürlich noch nicht den jeweiligen Lösungsalgorithmus.

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. Idealer weise wird ein Snake-Spiel zuerst gelöst die Daten erhoben und anschließend kann das Spiel jederzeit visuell aus den Daten rekonstruiert werden. Auf diese Weise könnte man unverfälschte Benchmarks erstellen. Allerdings müssten dafür jeder Schritt oder jeder erstellte Plan gespeichert werden. Bei bis zu 3 Millionen Schritten habe ich mich entschieden entweder Benchmarks zu erstellen oder das Spiel zu visualisieren. Nicht beides gleichzeitig.



Wenn die Schlange einem Hamiltonkreis folgt wird dieser mit einer blauen Linie dargestellt. Dies Ermöglicht auch zu sehen wenn Abkürzungen genommen werden, das heißt die Schlange den sicheren Pfad verlässt.

Algorithmen

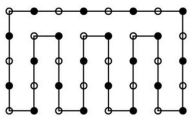
Sobald die Schnittstelle des Spiel einen Pfad anfragt. Gibt der ausgewählte Algorithmus einen Pfad und die benötigte Rechenzeit zurück. Die Schnittstelle Testet dann ob der Algorithmus einen freien und regelkonformen Pfad angibt. In den Daten werden dann die Benchmark-werte des Algorithmus gespeichert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

Allgemeine Komplexität

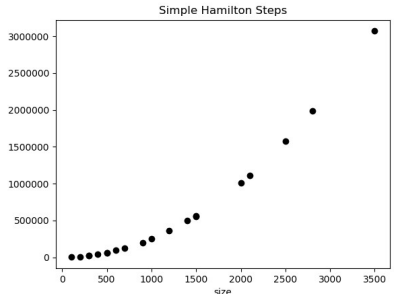
Grundsätzlich ist es Möglich ein Hamiltonkreis in linearer Zeit zu generieren. Für ein einfaches Beispiel, wird hier der einfache Hamiltonkreis genutzt (siehe Kapitel). Wobei es Grundsätzlich auch möglich ist es auch ein pseudozufälligen Hamiltonkreis in linearer Zeit zu generieren. Dies ist aber nur möglich, wenn es sich um ein einfaches Gitternetz handelt. Sobald die Schlange Wege blockiert scheint dies nicht mehr möglich. Zumindest ist das überprüfen von Hamiltonicity in einem Graphen grundsätzlich NP-Schwer.

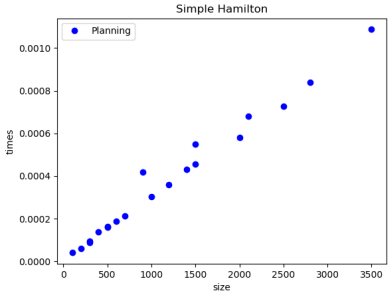
Idealer weise werden Abkürzungen zum Apfel nur genommen, wenn die Hamiltonicity im verbleibenden Graphen nachgewiesen werden kann. Aufgrund der Komplexität ist die hier implementierte Abkürzung optimistisch, aber nicht 100% zuverlässig.

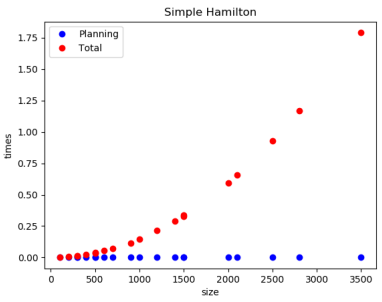
Einfacher Hamiltonkreis



Der einfache hamiltonkreis-Algorithmus generiert einen Pfad über das gesamte Spielfeld nach dem im Bild gegebenen Muster. Die Schlange läuft diesen Pfad ab bis das Spiel gewonnen ist. In den Ergebnissen ist zu sehen das diese Variante sehr schnell ist (blaue Punkte) und nur linear zur Größe Zeit benötigt. Die Benötigten Schritte (schwarz) um das Spiel zu beenden wächst aber Exponentiell und damit auch der backend-Overhead (Rot), also die benötigte Zeit um all diese Schritte auszuführen.

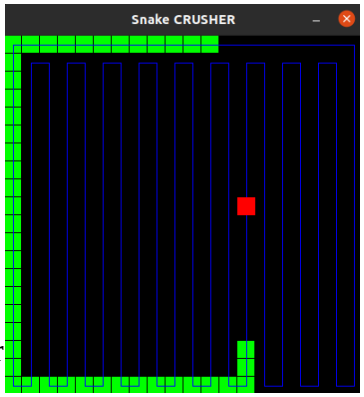




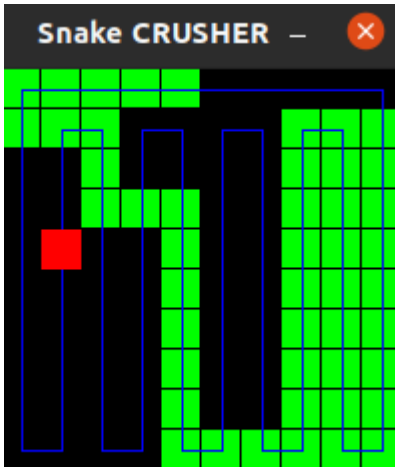


Abkürzungen

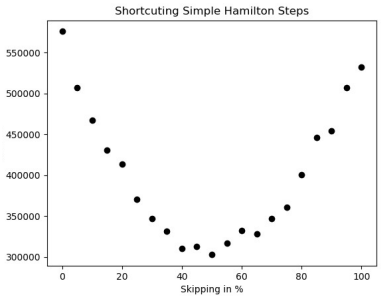
Für die Implementierung der Abkürzungen wird erst ein Hamiltonkreis generiert, dem die Schlange folgen soll. Sobald ein neuer Apfel generiert wird, wird der Pfad zum Apfel überprüft.

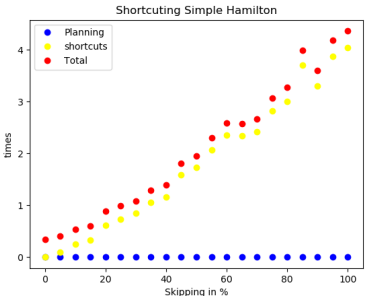


Es wird getestet ob der Kopf (nach rechts) springen kann und danach der Kopf näher am Apfel ist. Wenn eine Abkürzung gefunden wird werden die übersprungenen Felder aus dem Pfad gelöscht. Dies passiert aber nur, wenn genügend Sicherheitsabstand zum Schwanzende gewährleistet wird, da Äpfel sonst auf dem verbleibenden Weg auftauchen könnten und so einen Crash verursachen würden. Die Einschränkung das die Schlange nur Wege nach Recht überprüft wurde gemacht da dies die einzige sinnvollste Richtung im einfachen Hamitonkreis ist (siehe Bild) und die Rechenzeit überschaubarer gehalten wird. Dies könnte aber relativ leicht erweitert werden.



Ein Problem das bei diesem Ansatz entsteht ist, das bei einer genügend langen Schlange der Apfel immer öfter im übersprungenen Bereich auftaucht (siehe Bild unten). Daher überspringt die Schlange keinen Felder mehr, wenn ein bestimmte Prozentzahl des Spielfeldes aus Schlange besteht. Dennoch kann dieser Ansatz bis zu 50% der Schritte einsparen um das Spiel zu beenden (siehe Grafik links). Ergebnisse sind in Abhängigkeit der Prozentangabe geben, bis zu dessen Punkt die Schlange abkürzt.





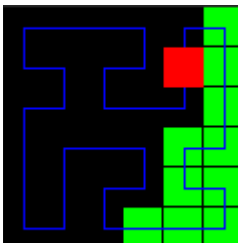
Tiefensuche

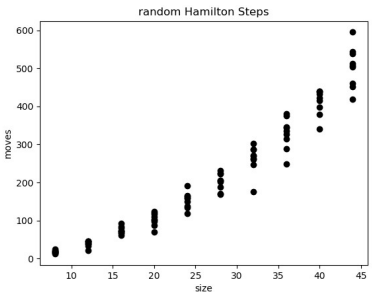
Pseudocode für normale Tiefensuche:

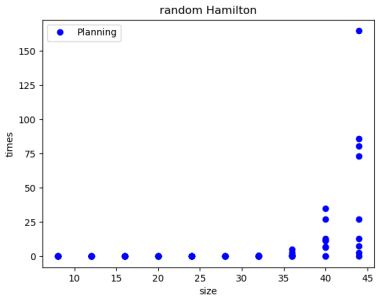
```
Depth-First-Search-Kickoff( Maze m )
    Depth-First-Search( m.StartCell )
End procedure

Depth-First-Search( MazeCell c )
    If c is the goal
        Exit
    Else
        Mark c "Visit In Progress"
        Foreach neighbor n of c
            If n "Unvisited"
                Depth-First-Search( n )
        Mark c "Visited"
End procedure
```

https://courses.cs.washington.edu/courses/cse326/03su/homework/hw3/dfs.html

Diese Tiefensuche lässt sich einfach Implementieren. Allerdings ist diese keine Lösung für das Problem, da sie nur ein Pfad zum Ziel findet, aber nicht garantiert, das alle Felder ablaufen werden. Der Algorithmus muss so angepasst werden, das der Zielknoten (also der Startpunkt) erreicht sein muss und die Länge des Pfades das gesamte Feld ausfüllt. Außerdem muss die Markierung von besuchten Feldern gelöscht werden können, da auch der Fad zum besuchten Feld entscheidend ist. Das erhöht die Komplexität deutlich. Außerdem haben so erzeugte Hamiltonkreis grundsätzlichen kein Vorteil bei der Schrittzahl.





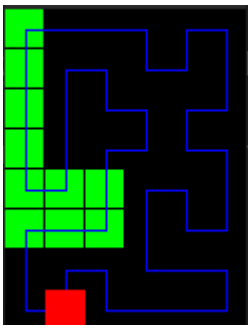


Die Benchmarks der Generierung zeigen das die im worst-Case benötigte Zeit mit der der Größe des Spielfeldes exponentiell wächst. Allerdings hat man auch oft mehr Glück und ein Ergebnis wird relativ schnell gefunden. Die schlechtesten Ergebnisse werden erreicht, wenn bei der Tiefensuche früh ein Fehler gemacht wird.

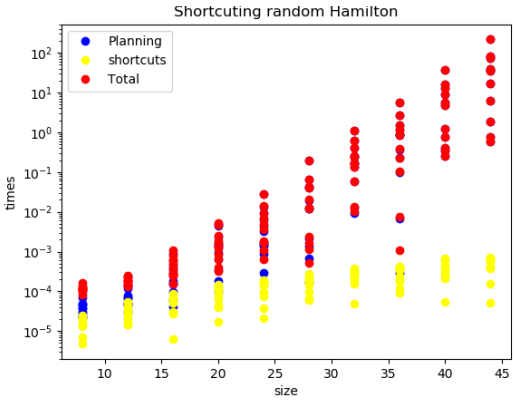
Vielleicht könnte die Suche in unterschiedliche Richtungen gestartet und so optimiert werden. Aktuell ist diese Suche so zu Zeitintensiv.

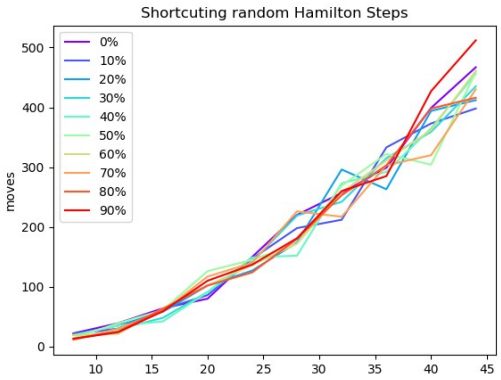
Ein Vorteil dieser Variante ist die Möglichkeit, ein Teil des Weges vorzugeben. Damit kann der Körper der Schlange und und ein kürzester Weg zum Apfel gegeben sein und dieser Algorithmus findet dann garantiert den passenden Hamiltonkreis bis zum Ende der Schlange, sofern er existiert. Falls nicht kann ein anderer Weg zum Apfel überprüft werden oder eine andere Strategie gewählt werden. Natürlich ist dies bei großen Spielfeldern nicht in annehmbaren Zeiten möglich.

Abkürzungen in Tiefensuche



Abkürzungen in einem Zufällig generierten Hamiltonkreis funktionieren weniger gut, da in der Aktuellen Implementation nur Wände nach rechts übersprungen werden können. Dies erspart einige Rechenkomplexität da 3 mal weniger Pfade berechnet werden müssen. Auch wenn es im einfachen Hamiltonkreis ein überspringen nach rechts sinnvoll ist. Ist diese Strategie in dieser Variante weniger überzeugend. Dennoch sieht man das Abkürzungen auch in dieser From noch einen kleinen Vorteil einbringen kann.





(% Angabe bis zu welcher Länge abgekürzt wird)

Answer Set Programming

Für die Generierung eines Pfades des Snake Spiels habe ich auch das Clingo System getestet, das auf Answer Set Programming (ASP) basiert und es Benutzern ermöglicht, ASP-Programme zu schreiben, auszuführen und zu debuggen.

Answer Set Programming ist eine deklarative Programmiersprache für das Lösen von Problemen der Wissensrepräsentation und des logischen Schließens in der Künstlichen Intelligenz genutzt wird.

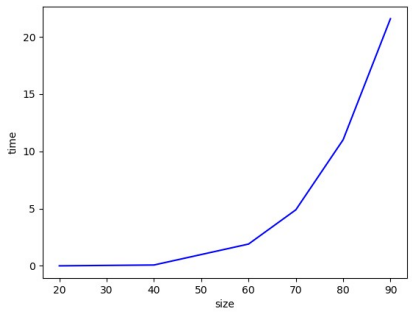
ASP-Programme bestehen aus Fakten und Regeln, die in eine bestimmte Syntax geschrieben sind. Die Fakten sind Aussagen über die Welt, die in der Regel als Atomare Aussagen dargestellt werden. Regeln geben an, welche Schlussfolgerungen aus den gegebenen Fakten gezogen werden können.

Eine besondere Eigenschaft von ASP ist, dass es verschiedene Arten von Negation unterstützt, einschließlich der Negation als Ausfall (Negation as Failure), die es ermöglicht, Aussagen als wahr anzunehmen, wenn sie nicht widerlegt werden können. Das Ziel von ASP ist es, eine Menge von Antwortmengen zu finden, die eine bestimmte Bedingung erfüllen. Eine Antwortmenge ist eine Menge von Aussagen, die alle wahr sind und zusammen eine konsistente Lösung des Problems darstellen.

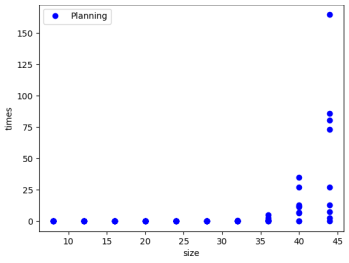
Die Clingo-API bietet eine Schnittstelle für Python-Entwickler, um Clingo-Programme in Anwendungen zu nutzen, steuern und auf die Ergebnisse zuzugreifen, was die Integration von ASP in andere Anwendungen ermöglicht.

Mit einem Logikprogramm lässt sich das Problem beschreiben und dann von Clingo Lösen. In ersten Tests schnitt diese Variante schon deutlich Effektiver ab als die Tiefensuche. Die

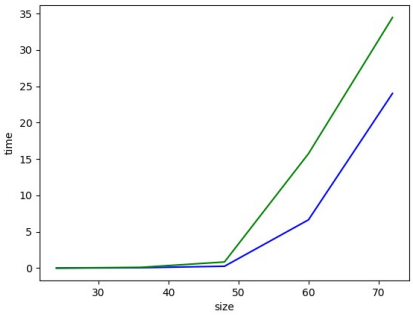
benötigte Zeit wächst dennoch Exponentiell zur Größe und ist damit bei größeren Spielfeldern immer noch übermäßig.
Im linken Bild ist eine Implementierung zu sehen die bei doppelt so großen Spielfeldern noch deutlich schneller ist, als die Tiefensuche.



ASP Implementation



Vergleich Tiefensuche



Zweite ASP Implementation

https://www.cs.utexas.edu/users/vl/papers/wiasp.pdf

Die Zweite ASP Implementation wird mit einer zweiten (grünen) Linie gezeigt. Diese Linie ist die Zeit die Clingo braucht denjenigen Hamiltonkreis zu finden mit dem kleinsten Weg zum Apfel. Dies wird mit folgender Ergänzung gelöst, die den Zeitpunkt T minimiert an dem das Essen erreicht wird:

```
#minimize{T:food(T)}.
```

Auch Wenn diese zweite Variante nicht ganz so schnell ist wie die erste, zeigt sie doch mit wie wenig Mehraufwand eine Optimierung zum Essen gefunden werden kann. Die ist natürlich nur die Rechenzeit für die Optimierung zum erste Apfel, aber diese Variante kann so erweitert werden, das der Pfad mit jedem Apfel neu optimiert wird.

Weitere Möglichkeiten

Eine Variante von John Tapsell nutzt Prim‘s Algorithmus um ein pseudozufälligen Hamiltonkreis zu finden. den minimalen Spannbaum eines Graphen zu finden. Wenn man den Algorithmus jedoch auf ein Gitternetz anwendet, kann man damit auch ein zufälliges Labyrinth generieren.

Das Verfahren besteht aus den folgenden Schritten:

1. Wähle einen zufälligen Startpunkt im Gitternetz und füge diesen in die Liste der besuchten Knoten ein.
2. Finde alle Nachbarn des aktuellen Knotens, die noch nicht in der Liste der besuchten Knoten sind und füge diese in eine Prioritätswarteschlange ein. Die Priorität der Knoten in der Warteschlange wird durch die Kosten der Kanten bestimmt, die den Knoten mit bereits besuchten Knoten verbinden. Die Kosten können zum Beispiel zufällig gewählt werden.

3. Wähle den Knoten mit der niedrigsten Priorität aus der Warteschlange und füge ihn in die Liste der besuchten Knoten ein. Füge auch die Kante hinzu, die den aktuellen Knoten mit dem ausgewählten Nachbarknoten verbindet, in eine Liste der Kanten ein, die das Labyrinth bilden.
4. Wiederhole Schritt 2 und 3, bis alle Knoten im Gitternetz besucht wurden.

Das Ergebnis ist ein minimales Spannbaum des Gitternetzes, das ein zufälliges Labyrinth bildet. Der minimale Spannbaum wird dadurch erreicht, dass der Algorithmus nur Kanten mit den niedrigsten Kosten hinzufügt und keine Schleifen im Graphen erzeugt.

Indem man den Startpunkt und die Kosten der Kanten variiert, kann man verschiedene Labyrinthe mit unterschiedlichen Schwierigkeitsgraden generieren.

Wenn man nun ein Labyrinth der halben Größe des Spielfeldes generiert uns dieses immer an der linken Wand entlang geht (die Gänge immer 2 Felder groß sind), dann kann man ein zufällig wirkenden Hamiltonkreis generieren. Diese Variante scheint in Linearer Zeit zu laufen und ist damit deutlich Effizienter als meine bisherigen Implementationen.

https://johnflux.com/2015/05/02/nokia-6110-part-3-algorithms/

Auf diesem Blog findet sich auch eine Möglichkeit Abkürzungen zu generieren. Im Rahmen dieser Arbeit habe ich es noch nicht geschafft, den C++ Code in Python zu übersetzen. Auch laufen Abkürzungen wie aus diesem Blog Gefahr, dass die neuen Äpfel ungünstig aufzutauchen und die Schlange doch noch in sich selbst fährt.

Meine eigene Implementation baut auf ein ähnliches Prinzip mit etwas mehr Sicherheitsabstand zum eigenen Ende. Dennoch ist aber theoretisch ein Crash auch möglich.

Schlusswort

Damit das Spiel mit 100% Sicherheit gelöst werden kann, muss immer ein vollständiger Hamiltonkreis garantiert sein. Auch wenn die Generierung von Hamiltonkreisen in linearer Zeit möglich ist, ist denjenigen Hamiltonkreis mit dem Kürzesten Pfad zum Apfel zu finden scheinbar ein NP-Vollständiges Problem. Zumindest ist das nachweisen von Hamiltonicity nachgewiesen NP schwer. Umso beeindruckender zeigt sich der Clingo-Solver, wenn in dieser Lösungsvariante die Darstellung des Problemraumes minimiert werden kann, könnten hier eventuell sogar noch bessere Ergebnisse erreicht werden. Auch scheint eine Kombination von hier vorgestellten Varianten eine gute Strategie. So könnten anfangs Abkürzungen genutzt werden, da sie dort besonders effektiv sind und anschließend mit Tiefensuche oder dem Clingo Solver die letzten Schritte optimiert werden, wenn das verbleibende freie Spielfeld klein genug ist.