Einleitung

-jeder von euch kennt vermutlich weltberühmte Sudoku-Rätsel, vor einigen Jahrzehnten in Japan erfunden

-> wie letztes Jahr beschäftigt 1. Runde des BWInf wieder mit japanischen Logikrätsel, jedoch nicht mit Sudoku, sondern mit den sogenannten Arukonen (weiter)

-was Arukonen sind und wie man sie erstellen kann werde ich euch in dieser kurzen Vorstellung näherbringen

-dafür werde ich zuerst mit meiner Gliederung starten (weiter)

Gliederung

-klassisch starten mit Erläuterung der Aufgabenstellung

2 Vorstellung der Lösungsidee

3 Umsetzung des Ansatzes

4 Verdeutlichung des Programms an Beispielen

1 Erläuterung der Aufgabenstellung

-Arukone-Rätsel besteht aus Gitter der Seitenlänge N, in welchem jedes Feld entweder eine Zahl enthält oder leer ist (weiter)

-jede Zahl kommt genau zweimal vor

-Aufgabe: jedes Paar gleicher Zahlen durch Linienzug zu verbinden (nur aus horizontalen und vertikalen Abschnitten, auch leere Felder erlaubt)

-> mögliche Lösung: weiter

-Herausforderung: solche Rätsel für ein gegebenes N mit N >= 4 zu generieren mit Bedingungen:

-sollen mindestens n/2 verschiedene Zahlen beinhalten

-nicht von Arukone-Solver gelöst werden können

2 Vorstellung der Lösungsidee

-schnell und ist intuitiv entstanden (weiter)

-> Ansatz: zufällig zwei leere Zellen bestimmen und versuchen, dies durch einen Linienzug zu verbinden

-> einfach dann für viele verschiedene Zellen machen und dadurch Rätsel erstellen

-für Pfadsuche in Gitter gibt ganz viele verschiedene Algorithmen:

-DFS, BFS, Backtracking oder auch A\*-Algorithmus (weiter)

-Breitensuche immer noch naheliegendste

- „überflutet“ jedoch Graphen und erforscht alle Wege, solange, bis Ziel gefunden

-kann entsprechend laufzeit- und speicherintensiv sein -> A\*-Algorithmus

-A\* basiert auf Heuristik (Näherung) -> wird genutzt um abzuschätzen, welche Zelle als nächstes zum Pfad hinzugefügt wird, um Endknoten möglichst schnell zu erreichen

-> nachdem ein paar verbunden, nächstes Paar usw.

-Pseudocode (weiter)

-Variable *num\_paths* initialisiert: Anzahl an schon gefundenen Pfaden darstellt

-maximale Anzahl an Pfaden wird ermittelt

-solange Anzahl an gefundenen Pfaden kleiner als max. Anzahl Pfade

-Gitter initialisiert

-jeweils zwischen freien Zellen Pfade gesucht

-Stichwort „gültig“ bedeutet, dass Pfad länger als 2 Zellen ist

-alles sehr theoretisch, deswegen noch einmal Erläuterung der Funktionsweise dieser Idee an Beispiel (weiter):

-Schritt 1: Zwei zufällige noch freie Punkte ausgewählt

-Schritt 2: diese werden durch A\* verbunden

-Schritt 3: wiederum zwei neue Punkte, welche wiederum verbunden werden

-Schritt 5 passiert dasselbe, wobei zwischen dieses jedoch kein möglicher Pfad

gefunden werden kann, ohne einen anderen Pfad zu übertreten

-Schritt 6: deswegen werden zwei neue Felder ausgewählt, zwischen welchen

die Pfadlänge 3 beträgt, also gerade so das Minimum erfüllt

3 Umsetzung

-Lösungsidee in Python: benutzerfreundlich ist und viele Bibliotheken anbietet

-Implementierung:

-Methode *initialize\_grid* -> 2D-Array der Größe N \* N (Nullen)

-Methode *get\_neighbours* -> Nachbarn einer Zelle (weiter)

-> Nachbarn der Zelle mit den Koordinaten (2, 1): (2, 0), (1, 1), (3, 1) und (2, 2)

-sollte natürlich auch für Zelle am Rand funktionieren, bspw. für (0, 1)

-> die entsprechenden Nachbarn wären dann (0, 0), (1, 1) und (0, 2)

-beiden Methoden wichtig für Herzstück des Programmes:

-A\*-Methode und ihrer Heuristik

-Heuristik (also zur Schätzung der Distanz von einer Zelle zur einer Zielzelle): (weiter)

Summe des Betrages der Differenz der x- und y-Koordinaten zweier Punkte

-> Manhattan-Distanz

-zur Implementierung von A\* verschiedene Datenstrukturen (weiter)

-Node: Instanz der Klasse namedtuple in Python, die grundlegend vier wichtige Sachen abspeichert:

-gesamten Kosten (bisherigen Kosten + Heuristik)

-Koordinaten der aktuellen Zelle

-bisherigen Kosten

-voraussichtlichen Kosten

-Set visited: speichert schon besuchte Zellen ab (weiter)

-Prioritätswarteschlange (sortiert nach Gesamtkosten): Zellen, die A\* abläuft (weiter)

-Dictionary came\_from: Speichert bisherigen Weg ab -> Rekonstruktion (weiter)

-Pseudocode: (weiter)

-als Parameter erhält er das Gitter, die Start- und die Zielzelle sowie N

-Datenstrukturen initialisiert

-solange Warteschlange nicht leer ist:

-nimm aktuelle Zelle runter

-wenn diese gleich der Zielzelle ist -> Rekonstruktion

-sonst füge sie zu den besuchten Zellen hinzu und schaue Nachbarn an

-wenn noch nicht besucht und noch frei:

-neue Zelle und füge sie in Queue ein

-> Zeitkomplexität (schwieriger als erwartet):

-Heuristik-Methode: O (1), da arithmetische Operationen (weiter)

-while-Schleife: wie oft kann sie maximal durchlaufen werden?

-> dabei muss bedacht werden, dass jede freie Nachbarzelle in die

Queue eingefügt wird

-Frage wäre also:

-wenn A\* also zahlreiche Umwege gehen muss, aber trotzdem

kürzesten Weg nimmt, was wäre dann die maximale Anzahl

der Elemente in Warteschlange

-> Mathe-Olympiade-Feeling, daher einfach b für maximale Anzahl (weiter)

-for-Schleife: durchläuft max. 4 Nachbarn, also O(1) (weiter)

-Nachbar schon in Warteschlange: O(b) (weiter)

-ich verwendete heappop und heappush um Elemente aus der Schlange zu entfernen oder hinzuzufügen: log(b) (weiter)

-multipliziert man alle Laufzeiten: O(b²log(b)) (weiter)

(weiter)

-ich hoffe, dass ich euch mit der Vorstellung meiner Lösung einen kurzen Einblick in die Welt der Arukone-Rätsel geben konnte und vielleicht, wenn ihr euren Opa das nächste Mal ein Sudoku-Rätsel lösen seht, könnt ihr ihm ja ein Arukone-Rätsel vorschlagen