Müllabfuhr ? A1 ♣ 61015 Leonhard Masche 110.04.2022 Inhaltsverzeichnis 1. Lösungsidee
1. Verbesserungen 2. Aufbau 2. Umsetzung 3. Beispiele 4. Quellcode Lösungsidee
Zuerst wird das Straßennetz in einer Adjacency-List Datenstruktur gespeichert. Nun wird von der Basis (Vertex 0) ausgehend ein modifizierter Breadth-First-Search Algorithmus verwendet, um die in Frage kommenden Routen zu erhalten. Der Algorithmus speichert die aktuelle Node zusammen mit dem Pfad der bis dorthin beschritten wurde in visited, und zeichnet alle Möglichkeiten von der derzeitigen Node weiterzugehen (rückwärtsgehen ausgeschlossen), zusammen mit der dann entstehenden Länge des Pfades auf. Wenn ein Pfad auf eine Vertex aus dem visited-Set trifft, wird der jetzige Pfad mit dem in visited gespeicherten Pfad zusammengefügt und als Ergebnis des Algorithmus in der Liste paths gespeichert. Wenn alle Straßen vom Algorithmus abgedeckt wurden, ist dieser Schritt fertig.
Do. Mo. Zeutrale Fr. Fr. Mi. Di.
Da es passieren kann, dass eine Straße ofters als benötigt 'befahren' wird, werden im nächsten Schritt unnötige Wege entfernt. Nun ist allerdings nicht garantiert, dass paths die richtige Länge (z.B. 5 Tage) hat. Deshalb werden, solange die Länge von paths größer als die Anzahl der Tage ist, die beiden kürzesten Pfade kombiniert. Somit wird eine zu lange Liste
verkürzt. Ist sie jedoch zu kurz, wird sie mit Nullen (Wagen bleibt in der Zentrale) aufgefüllt. Verbesserungen Nicht-Integer Gewichte Eine vorgenommene Verbesserung ist das Einlesen von Fließkommazahl-Gewichtungen der Straßen. Es ist unrealistisch dass in einem echten Szenario Straßen eine Länge von z.B. genau 480m haben. Um das zu implementieren wird der dritte Wert aus den Beispieldateien als float
Arbiträre Anzahl Tage Auch kann eine Anzahl an Tagen eingegeben werden, für die geplant werden soll. So kann zum Beipiel ein Fahrplan für zwei Wochen erstellt werden. Dazu werden einfach die merging- und padding-Schritte am Ende der get_paths Funktion angepasst. Aufbau
def remove_by_exp(exp: Callable[[Any], bool], lst: List) Entfernt das erste Element bei dem exp 'True' zurückgibt. program.py
class CityGraph Klasse die ein Straßennetz (ungerichteter gewichteter Graph) repräsentiert. definit(vertices: List[int], edges: List[Tuple[int, int, float]]) Initialisiert den CityGraph mit einer Liste der Vertices und der adjacency-list. @classmethod
def _from_bwinf_file(path: str) -> 'CityGraph' Liest eine Beispieldatei ein, und gibt einen CityGraph zurück. def _contains_all_edges(paths: Iterable[Iterable[int]]) -> bool Gibt als Wahrheitswert zurück, ob die gegebene Liste an Pfaden alle 'Straßen' im Graph abdeckt. def get_paths(days: int = 5) -> List[Tuple[float, Tuple[int,]]]
Gibt eine Liste zurück, die Tuples mit dem Pfad, und der Länge dessen an erster Stelle, enthält. Umsetzung Das Programm ist in der Sprache Python umgesetzt. Der Aufgabenordner enthält neben dieser Dokumentation eine ausführbare Python-Datei program.py. Diese Datei ist mit einer Python-Umgebung ab der Version 3.6 ausführbar. Wird das Programm gestartet, wird zuerst eine Eingabe in Form einer einstelligen Zahl erwartet, um ein bestimmtes Beispiel auszuwählen. (Das heißt: 0 für Beispiel muel Labfuhr 0. txt).
Dann wird nach der Anzahl der zu planenden Tage gefragt (default ist 5). Nun wird die Logik des Programms angewandt und die Ausgabe erscheint in der Kommandozeile. Beispiele Hier wird das Programm auf die neun Beispiele aus dem Git-Repo, und ein eigenes angewendet:
muellabfuhr0.txt 10 13 0 2 1 0 4 1 0 6 1 0 8 1 : 5 6 1
6 7 1 7 8 1 8 1 1 8 9 1 Ausgabe zu muellabfuhr0.txt
Tag 1: 0 -> 8 -> 9 -> 8 -> 0, Gesamtlaenge: 4.0 Tag 2: 0 -> 6 -> 7 -> 8 -> 0, Gesamtlaenge: 4.0 Tag 3: 0 -> 4 -> 5 -> 6 -> 0, Gesamtlaenge: 4.0 Tag 4: 0 -> 2 -> 3 -> 4 -> 0, Gesamtlaenge: 4.0 Tag 5: 0 -> 2 -> 1 -> 8 -> 0, Gesamtlaenge: 4.0 Maximale Lange einer Tagestour: 4.0
muellabfuhr1.txt 8 13 0 4 6 0 5 6 0 6 1 1 3 9 :
3 6 1 4 5 5 4 7 8 5 7 2 6 7 1 Ausgabe zu muellabfuhr1.txt
Tag 1: 0 -> 6 -> 7 -> 5 -> 4 -> 7 -> 6 -> 0, Gesamtlaenge: 19.0 Tag 2: 0 -> 6 -> 3 -> 2 -> 3 -> 6 -> 0, Gesamtlaenge: 18.0 Tag 3: 0 -> 4 -> 3 -> 6 -> 0, Gesamtlaenge: 15.0 Tag 4: 0 -> 6 -> 1 -> 3 -> 6 -> 0, Gesamtlaenge: 13.0 Tag 5: 0 -> 6 -> 3 -> 5 -> 0, Gesamtlaenge: 11.0 Maximale Lange einer Tagestour: 19.0
muellabfuhr2.txt 15 34 0 5 1 0 6 1 0 9 1 1 6 1 :
9 10 1 9 12 1 9 13 1 10 14 1 13 14 1 Ausgabe zu muellabfuhr2.txt
Tag 1: 0 -> 9 -> 13 -> 14 -> 5 -> 0 -> 5 -> 11 -> 3 -> 4 -> 6 -> 0 -> 5 -> 11 -> 8 -> 14 -> 5 -> 0 -> 5 -> 14 -> 2 -> 10 -> 9 -> 0, Gesamtlaenge: 23.0 Tag 2: 0 -> 5 -> 14 -> 7 -> 1 -> 6 -> 0 -> 5 -> 14 -> 8 -> 7 -> 9 -> 0, Gesamtlaenge: 12.0 Tag 3: 0 -> 5 -> 14 -> 10 -> 4 -> 6 -> 0 -> 6 -> 1 -> 13 -> 4 -> 6 -> 0, Gesamtlaenge: 12.0 Tag 4: 0 -> 6 -> 4 -> 3 -> 13 -> 9 -> 0 -> 9 -> 12 -> 8 -> 2 -> 11 -> 5 -> 0, Gesamtlaenge: 13.0 Tag 5: 0 -> 5 -> 9 -> 6 -> 0 -> 5 -> 14 -> 6 -> 0 -> 9 -> 7 -> 11 -> 5 -> 0 -> 9 -> 12 -> 1 -> 6 -> 0, Gesamtlaenge: 18.0 Maximale Lange einer Tagestour: 23.0
muellabfuhr3.txt 15 105 0 1 1 0 2 1 0 3 1 0 4 1
: 11 13 1 11 14 1 12 13 1 12 14 1 13 14 1 Ausgabe zu muellabfuhr3.txt
Tag 1: 0 -> 6 -> 1 -> 0 -> 8 -> 1 -> 0 -> 10 -> 1 -> 0 -> 12 -> 1 -> 0 -> 14 -> 1 -> 0 -> 1 -> 3 -> 2 -> 0 -> 1 -> 5 -> 2 -> 0 -> 1 -> 7 -> 2 -> 0 -> 1 -> 9 -> 2 -> 0 -> 1 -> 11 -> 2 -> 0 -> 1 -> 13 -> 2 -> 0 -> 2 -> 4 -> 3 -> 0 -> 2 -> 6 -> 3 -> 0 -> 2 -> 8 -> 3 -> 0 -> 2 -> 10 -> 3 -> 0 -> 2 -> 12 -> 3 -> 0, Gesamtlaenge: 59.0 Tag 2: 0 -> 2 -> 14 -> 3 -> 0 -> 3 -> 5 -> 4 -> 0 -> 3 -> 7 -> 4 -> 0 -> 3 -> 9 -> 4 -> 0 -> 3 -> 11 -> 4 -> 0 -> 3 -> 13 -> 4 -> 0 -> 4 -> 6 -> 5 -> 0 -> 4 -> 8 -> 5 -> 0, Gesamtlaenge: 32.0 Tag 3: 0 -> 4 -> 10 -> 5 -> 0 -> 4 -> 12 -> 5 -> 0 -> 4 -> 14 -> 5 -> 0 -> 5 -> 7 -> 6 -> 0 -> 5 -> 9 -> 6 -> 0 -> 5 -> 11 -> 6 -> 0 -> 5 -> 13 -> 6 -> 0 -> 6 -> 8 -> 7 -> 0, Gesamtlaenge: 32.0 Tag 4: 0 -> 6 -> 10 -> 7 -> 0 -> 6 -> 12 -> 7 -> 0 -> 6 -> 14 -> 7 -> 0 -> 7 -> 9 -> 8 -> 0 -> 7 -> 11 -> 8 -> 0 -> 9 -> 9 -> 9 -> 9 -> 9
13 -> 8 -> 0 -> 8 -> 10 -> 9 -> 0 -> 8 -> 12 -> 9 -> 0, Gesamtlaenge: 32.0 Tag 5: 0 -> 8 -> 14 -> 9 -> 0 -> 9 -> 11 -> 10 -> 0 -> 9 -> 13 -> 10 -> 0 -> 12 -> 11 -> 0 -> 14 -> 11 -> 0 -> 11 -> 13 -> 12 -> 0 -> 12 -> 14 -> 13 -> 0 -> 2 -> 1 -> 0 -> 4 -> 1 -> 0, Gesamtlaenge: 34.0 Maximale Lange einer Tagestour: 59.0 muellabfuhr4.txt
10 10 0 1 1 0 9 1 1 2 1 2 3 1 3 4 1 4 5 1 5 6 1
6 7 1 7 8 1 8 9 1 Ausgabe zu muellabfuhr4.txt Tag 1: 0 -> 1 -> 2 -> 3 -> 4 -> 5 -> 6 -> 7 -> 8 -> 9 -> 0, Gesamtlaenge: 10.0
Tag 2: 0, Gesamtlaenge: 0.0 Tag 3: 0, Gesamtlaenge: 0.0 Tag 4: 0, Gesamtlaenge: 0.0 Tag 5: 0, Gesamtlaenge: 0.0 Maximale Lange einer Tagestour: 10.0 muellabfuhr5.txt
50 989 0 2 8 0 3 12 0 5 6 0 6 9 : 46 48 8 46 49 1
47 48 7 47 49 6 48 49 9 Ausgabe zu muellabfuhr5.txt Keine Pfade gefunden! (Mehrere unverbundene Straßennetze). (pop from empty list) Maximale Lange einer Tagestour: 0
muellabfuhr6.txt 100 204 0 4 7782 0 44 5495
1 3 4633 2 5 18959 .: 98 32 11629 98 35 1367 98 45 11403 99 27 4355 99 43 3450
Ausgabe zu muellabfuhr6.txt Keine Pfade gefunden! (Mehrere unverbundene Straßennetze). (pop from empty list) Maximale Lange einer Tagestour: 0
muellabfuhr7.txt 500 1636 0 317 164 1 0 48298 1 4 70 1 250 133 .: 498 353 68
499 15 45 499 125 56 499 129 111 499 365 55 Ausgabe zu muellabfuhr7.txt
Keine Pfade gefunden! (Mehrere unverbundene Straßennetze). (pop from empty list) Maximale Lange einer Tagestour: 0 muellabfuhr8.txt 1000 3453
0 294 3093 0 303 3855 0 420 2567 0 593 4699 : 998 696 2946 998 813 4484 999 417 6731 999 843 1857 999 247 3574
Ausgabe zu muellabfuhr8.txt Keine Pfade gefunden! (Mehrere unverbundene Straßennetze). (pop from empty list) Maximale Lange einer Tagestour: 0
muellabfuhr9.txt 3 3 0 1 0.5 0 2 0.5 1 2 50.7 eigenes Beispiel zur Demonstration von float-Gewichten
Ausgabe zu muellabfuhr9.txt Tag 1: 0 -> 2 -> 1 -> 0, Gesamtlaenge: 51.7 Tag 2: 0, Gesamtlaenge: 0.0 Tag 3: 0, Gesamtlaenge: 0.0 Tag 4: 0, Gesamtlaenge: 0.0
Tag 5: 0, Gesamtlaenge: 0.0 Maximale Lange einer Tagestour: 51.7 Quellcode utility.py from typing import Any, Callable, List
<pre>def remove_by_exp(exp: Callable[[Any], bool], lst: List): for i in lst: try: if exp(i): lst.remove(i)</pre>
pass program.py
<pre>from collections import Counter from os.path import dirname, join from typing import (FrozenSet, Iterable, List, Mapping, Set,</pre>
<pre>"""A class representing the city graph.""" vertices: Mapping[int, Mapping[int, float]] # {vertex_id: {connected_vertex_id: distance},} edgeset: Set[FrozenSet[int]] # {{vertex_id, vertex_id}, {vertex_id, vertex_id},} @classmethod def _from_bwinf_file(cls, path: str) -> 'CityGraph': """ Load the CityGraph from an bwinf example file.</pre> Parameters
<pre>Parameters path : str The path to the bwinf file. """ with open(path, 'r') as f: lines = f.read().split('\n') n, m = map(int, lines[0].split()) return cls(</pre>
<pre>return cls(list(range(n)), [(int(v), int(u), float(length)) for v, u, length in [line.split() for line in lines[1: m + 1]]]) definit(self, vertices: List[int], edges: List[Tuple[int, int, float]]): self.vertices = {v: {} for v in vertices} self.edgeset = set()</pre>
<pre>for edge in edges: v, u, len_ = edge self.edgeset.add(frozenset((v, u))) self.vertices[v][u] = len_ self.vertices[u][v] = len_ def _contains_all_edges(self, paths: Iterable[Iterable[int]]) -> bool: # convert all paths into an edgesets flattened = tuple(j for path in paths for j in path) edges = set(frozenset((flattened[i], flattened[i+1])) for i in range(len(flattened)-1))</pre>
<pre>edges = set(frozenset((flattened[i], flattened[i+1])) for i in range(len(flattened)-1)) # check if all edges are contained in the resulting edgeset for edge in self.edgeset: if edge not in edges: return False return True def get_paths(self, days: int = 5) -> List[Tuple[float, Tuple[int,]]]: # get paths using bfs-type algorithm visited: Mapping[int, Tuple[float, List[int]]] = {} # {visited_node_id: (length, path)}</pre>
<pre>paths: List[Tuple[float, Tuple[int,]]] = [] # [(length, (path)),] queue: List[Tuple[float, Tuple[int,]]] = [(0.0, [0])] # [(distance, path),] try: while not selfcontains_all_edges(map(lambda x: x[1], paths)): queue.sort() current_length, current_path = queue.pop(0) if current_path[-1] in visited: # check if path meets another path TODO make path not meet with itself paths.append((current_length+visited[current_path[-1]][0],</pre>
<pre>(*visited[current_path[-1]][1], *reversed(current_path[:-1])))) visited[current_path[-1]] = (current_length, current_path) # remove the path merging into the current path remove_by_exp(lambda x: x[1][-1] == current_path[-2], queue) continue if len(self.vertices[current_path[-1]]) == 1: # check if it's a dead end paths.append((current_length*2, (*current_path, *reversed(current_path[:-1])))) continue</pre>
<pre>continue visited[current_path[-1]] = (current_length, current_path) for next_node_id in self.vertices[current_path[-1]]: if next_node_id == (current_path[-2] if len(current_path) > 1 else None): # skip going backwards</pre>
<pre># remove unneeded paths paths.sort(reverse=True) edgecounts = Counter(frozenset((path[i], path[i+1])) for _, path in paths for i in range(len(path)-1)) keys = edgecounts.keys() for len_, path in paths: edgecount = Counter(frozenset((path[i], path[i+1])) for i in range(len(path)-1)) tmp = edgecounts-edgecount if not any(v < 1 for v in tmp.values()) and tmp.keys() == keys:</pre>
<pre>if not any(v < 1 for v in tmp.values()) and tmp.keys() == keys: paths.remove((len_, path)) edgecounts.subtract(edgecount) # merge paths while they are > target_n_days while len(paths) > days: paths.sort() first = paths.pop(0) second = paths[0] paths[0] = (first[0] + second[0], (*first[1], *second[1][1:]))</pre>
<pre># pad to length of target_n_days while len(paths) < days: paths.append((0.0, (0,))) return paths # repl while True:</pre>
<pre>try: pth = join(dirname(file),</pre>
<pre>print(f'Maximale Lange einer Tagestour: {maxlen}') except Exception as e:</pre>