Müllabfuhr

? A1 👤 61015 Leonhard Masche 10.04.2022

Inhaltsverzeichnis

Lösungsidee

Zuerst ist bei dieser Problemstellung eine gewisse Ähnlichkeit zum `min-max k-Chinese Postman problem` festzustellen. Da dieses Problem NP-Schwer ist, wird hier ein meta-heuristischer Algorithmus zur Annäherung an eine möglichst optimale Lösung verwendet, wie in dieser Arbeit beschrieben. Als Startpunkt wird ein einziger Pfad durch den Graph verlegt (vgl. Hierholzer's algorithm) und die restlichen Tage mit Nullen aufgefüllt.
Nun wird optimiert: Kurzgesagt wird iterativ eine 'Nachbarschaft', also leichte Veränderungen zweier Touren durch das Verschieben von zwei Kanten berechnet, und der beste Kandidat, der nicht in der tabu-Liste enthalten ist, wird weiterverbessert. Wenn zwei Möglichkeiten zur Weiterverbesserung gleich 'gut' sind, wird zufällig eine der beiden ausgewählt. Somit ist der Algorithmus zwar nicht deterministisch, mit ausreichender Laufzeit wir die Menge der möglichen Ergebnisse jedoch immer enger. Diese Auswahl wird nun der tabu-Liste hinzugefügt. Die tabu-Liste hat eine bestimmte Zeit, für die Elemente nicht noch einmal ausgewählt werden dürfen. Diese Zeit hat mit einem Wert von `20` (20 Iterationen) gute Ergebnisse geliefert. Zusätzlich wird der Algorithmus durch ein Limit von `100` Iterationen ohne Verbesserung, und eine maximale Laufzeit von `600` Sekunden beschränkt.

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
eingelesen.

Home-office

Eine Funktionalität des Verbesserungs-algorithmus, die im Aufsatz nicht beschrieben wurde, ist die Möglichkeit, in der Zentrale zu bleiben. Dies ist der Fall, wenn Pfade nicht mehr sinnvoll verkürzt werden können, und somit das Ausfahren an zusätzlichen Tagen keine Verbesserung bringt. Der ausgegebene Pfad ist dann einfach `0`.

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. Diese Veränderung and `k` wird auch in der Optimisierungsphase berücksichtigt.

Dropout

Das erste was die Optimierung macht ist, die Pfade gerecht aufzuteilen. Hierfür reicht es, nicht jede Möglichkeit zu betrachten. Dafür gibt es den dropout-Wert. Am Anfang werden z.B. 10% aller Möglichkeiten ignoriert. Mit `dropout_fn` verändert sich dieser Wert über die Laufzeit des Algorithmus und somit wird die Auswahl am Ende der Optimierung 'feiner'.

Aufbau

utility.py

class TabuList

> Eine tabu-Liste, die Elemente für eine Zeit von `tenure` als tabu markiert.

def TabuList.__init__(default_tenure: int, cleanup_freq: int = 20)

> Initialisiert die tabu-Liste mit einer default_tenure.

def TabuList._cleanup()

> Interne Methode. Wird aufgerufen, um abgelaufene Einträge aus der Liste zu löschen.

def TabuList.add(item: Hashable, tenure: int = None)

> Setzt ein `item` für eine Zeit von `tenure or default_tenure` auf die tabu-Liste.

def TabuList.get(item: Hashable) -> int

> Gibt die Zeit zurück, bis `item` nicht mehr tabu ist. (Sonst `0`)

def TabuList.tick()

> Inkrementiert die Zeit um eins.

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.

def CityGraph.__init__(vertices: List[int], edges: List[Tuple[int, int, float]])

> Initialisiert den CityGraph mit einer Liste der Vertices und der adjacency-list.

@classmethod
def CityGraph._from_bwinf_file(path: str) -> 'CityGraph'

> Liest eine Beispieldatei ein, und gibt einen CityGraph zurück.

def is_connected(self) -> bool

> Gibt als Wahrheitswert zurück, ob der Graph verbunden ist, d. h. es gibt nur ein verbundenes Straßennetz.

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.

tabu_optimization.py

def MMKCPP_TEE_TabuSearch(G: Dict[int, Dict[int, float]], k: int = 5, maxNOfItsWithoutImprovement: int = 100, maxRunningTime: float = 0, dropout: float = 0.1, dropout_fn: Callable = lambda x: x1.2, tabuTenure: int = 20) -> List[Tuple[int, ...]]**

> Min-Max K-Chinese Postman Problem - Two Edge Exchange. Findet und verbessert `k` Touren, die alle Kanten im Graph abdecken.
> `G` ist eine adjacency-List in der auch die Gewichte der Kanten gespeichert sind.
> `maxNOfItsWithoutImprovement` ist die Maximalzahl der Iterationen ohne das Finden einer besseren Lösung, dass der Algorithmus abgebrochen wird.
> `maxRunningTime` ist die maximale Laufzeit des Algorithmus, bevor dieser abgebrochen wird.
> `tabuTenure` ist die Anzahl an Iterationen, die ein schon besuchtes Element als tabu markiert wird.

die folgenden Funktionen befinden sich innerhalb der Funktion MMKCPP_TEE_TabuSearch

def edges(tour: Tuple[int, ...]) -> Iterable[set]

> Gibt alle im Pfad `tour` enthaltenen Kanten in der Form `[{0, 1}, {1, 2}, ...]` zurück.

def w_tour(tour: Tuple[int, ...]) -> float

> Gibt die Länge eines Pfades `tour` zurück.

def w_max_tours(tours: Iterable[Tuple[int, ...]]) -> float

> Gibt die Länge aller Pfade in `tours` zurück. Dies ist auch die cost-Funktion, die es zu minimieren gilt.

def edgecount_tour(tour: Tuple[int, ...]) -> collections.Counter

> Zählt alle Kanten im Pfad `tour`.

def edgecount_tours(tours: List[Tuple[int, ...]]) -> collections.Counter

> Zahlt alle Kanten in den Pfaden `tours`.

def MergeWalkWithTour(tour: Tuple[int, ...], walk: Tuple[int, ...]) -> Tuple[int, ...]

> Verbindet `walk` (2 Kanten) mit dem jetzigen Pfad `tour`.

def SeparateWalkFromTour(tour: Tuple[int, ...], walk: Tuple[int, ...]) -> Tuple[int, ...]

> Entfernt die Kanten `walk` im Pfad `tour`, während aufgepasst wird, dass der Pfad verbunden bleibt.

def ReorderToClosedWalk(edgeset: List[set]) -> Tuple[int, ...]

> Ordnet die Kanten `edges` so, dass ein geschlossener Pfad, der bei Kreuzung 0 anfängt und endet, entsteht.

def RemoveEvenRedundantEdges(tour: Tuple[int, ...], tours: List[Tuple[int, ...]]) -> Tuple[int, ...]

> Entfernt Kanten im Pfad `tour`, die zu gerader Zahl vorkommen, immernoch in anderen Touren vorkommen, und den Pfad verbunden lassen.

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 `muellabfuhr0.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 (kann bis zu 10 Minuten dauern!).

Beispiele

Hier wird das Programm auf die neun Beispiele aus dem Git-Repo, und ein eigenes angewendet (jeweils mit einem Ziel von `5` Tagen):

(die Anzahl der Punkte ist die Anzahl der vorgenommenen Verbesserungen)

`muellabfuhr0.txt`

dropout 0.1

```
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 -> 7 -> 6 -> 0, Gesamtlaenge: 4.0
Tag 2: 0 -> 6 -> 5 -> 4 -> 0, Gesamtlaenge: 4.0
Tag 3: 0 -> 4 -> 3 -> 2 -> 0, Gesamtlaenge: 4.0
Tag 4: 0 -> 8 -> 9 -> 8 -> 0, Gesamtlaenge: 4.0
Tag 5: 0 -> 8 -> 1 -> 2 -> 0, Gesamtlaenge: 4.0
Maximale Lange einer Tagestour: 4.0
```

`muellabfuhr1.txt` - *dropout 0.1*

```
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 -> 3 -> 2 -> 3 -> 6 -> 0, Gesamtlaenge: 18.0
Tag 2: 0 -> 6 -> 7 -> 4 -> 0 -> 6 -> 0, Gesamtlaenge: 18.0
Tag 3: 0 -> 6 -> 1 -> 3 -> 6 -> 0, Gesamtlaenge: 13.0
Tag 4: 0 -> 5 -> 3 -> 6 -> 0, Gesamtlaenge: 11.0
Tag 5: 0 -> 6 -> 7 -> 5 -> 4 -> 3 -> 6 -> 0, Gesamtlaenge: 18.0
Maximale Lange einer Tagestour: 18.0
```

`muellabfuhr2.txt` - *dropout 0.1*

```
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 -> 10 -> 2 -> 8 -> 11 -> 3 -> 4 -> 3 -> 13 -> 9 -> 0, Gesamtlaenge: 11.0
Tag 2: 0 -> 9 -> 13 -> 14 -> 10 -> 14 -> 7 -> 8 -> 12 -> 9 -> 6 -> 0, Gesamtlaenge: 11.0
Tag 3: 0 -> 6 -> 14 -> 13 -> 9 -> 5 -> 14 -> 6 -> 1 -> 7 -> 9 -> 0, Gesamtlaenge: 11.0
Tag 4: 0 -> 5 -> 11 -> 2 -> 14 -> 8 -> 12 -> 1 -> 7 -> 9 -> 0, Gesamtlaenge: 10.0
Tag 5: 0 -> 5 -> 11 -> 7 -> 1 -> 13 -> 4 -> 10 -> 4 -> 6 -> 0, Gesamtlaenge: 10.0
Maximale Lange einer Tagestour: 11.0
```

`muellabfuhr3.txt` - *dropout 0.1*

```
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 -> 11 -> 6 -> 10 -> 3 -> 12 -> 9 -> 11 -> 1 -> 10 -> 9 -> 6 -> 1 -> 3 -> 8 -> 14 -> 4 -> 10 -> 5 -> 2 -> 14 ->
0, Gesamtlaenge: 21.0
Tag 2: 0 -> 13 -> 5 -> 6 -> 3 -> 0 -> 2 -> 12 -> 14 -> 11 -> 3 -> 13 -> 4 -> 1 -> 12 -> 13 -> 10 -> 11 -> 13 -> 9 -> 7 ->
0, Gesamtlaenge: 21.0
Tag 3: 0 -> 6 -> 14 -> 3 -> 7 -> 5 -> 9 -> 0 -> 8 -> 2 -> 10 -> 14 -> 5 -> 11 -> 2 -> 4 -> 7 -> 10 -> 12 -> 5 -> 4 -> 0,
Gesamtlaenge: 21.0
Tag 4: 0 -> 12 -> 6 -> 13 -> 7 -> 11 -> 4 -> 9 -> 3 -> 4 -> 8 -> 1 -> 9 -> 14 -> 7 -> 8 -> 9 -> 2 -> 13 -> 14 -> 1 -> 0,
Gesamtlaenge: 21.0
Tag 5: 0 -> 10 -> 8 -> 13 -> 1 -> 2 -> 6 -> 8 -> 11 -> 12 -> 8 -> 5 -> 1 -> 7 -> 6 -> 4 -> 12 -> 7 -> 2 -> 3 -> 5 -> 0,
Gesamtlaenge: 21.0
Maximale Lange einer Tagestour: 21.0
```

`muellabfuhr4.txt` - *dropout 0.1*

```
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
Tag 3: 0, Gesamtlaenge: 0
Tag 4: 0, Gesamtlaenge: 0
Tag 5: 0, Gesamtlaenge: 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
```

Keine Ausgabe. (Iteration dauert zu lange)

`muellabfuhr6.txt` - *dropout 0.9*

```
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`

```
......
Tag 1: 0 -> 93 -> 35 -> 93 -> 4 -> 0 -> 58 -> 98 -> 45 -> 98 -> 10 -> 45 -> 93 -> 35 -> 98 -> 93 -> 58 -> 45 -> 42 -> 67
-> 42 -> 5 -> 86 -> 41 -> 86 -> 77 -> 67 -> 41 -> 90 -> 41 -> 42 -> 86 -> 72 -> 13 -> 77 -> 84 -> 2 -> 72 -> 2 -> 84 ->
13 -> 5 -> 2 -> 5 -> 67 -> 77 -> 84 -> 19 -> 39 -> 60 -> 39 -> 83 -> 28 -> 30 -> 28 -> 31 -> 28 -> 82 -> 31 -> 82 -> 28 -
> 83 -> 33 -> 83 -> 39 -> 30 -> 33 -> 60 -> 33 -> 30 -> 25 -> 30 -> 83 -> 25 -> 39 -> 19 -> 2 -> 13 -> 2 -> 19 -> 48 ->
25 -> 48 -> 10 -> 4 -> 44 -> 60 -> 61 -> 60 -> 59 -> 62 -> 54 -> 17 -> 9 -> 56 -> 9 -> 17 -> 56 -> 17 -> 22 -> 24 -> 61 -
> 24 -> 82 -> 70 -> 82 -> 24 -> 22 -> 47 -> 56 -> 26 -> 47 -> 9 -> 22 -> 54 -> 9 -> 26 -> 29 -> 53 -> 29 -> 26 -> 29 ->
57 -> 92 -> 79 -> 57 -> 79 -> 57 -> 53 -> 95 -> 53 -> 80 -> 65 -> 51 -> 81 -> 65 -> 80 -> 64 -> 63 -> 80 -> 76 -> 63 ->
80 -> 53 -> 80 -> 64 -> 51 -> 65 -> 76 -> 65 -> 81 -> 76 -> 80 -> 16 -> 92 -> 74 -> 16 -> 89 -> 96 -> 89 -> 40 -> 89 ->
99 -> 96 -> 99 -> 43 -> 99 -> 27 -> 99 -> 40 -> 66 -> 43 -> 27 -> 87 -> 27 -> 94 -> 15 -> 50 -> 7 -> 15 -> 12 -> 50 -> 85
-> 15 -> 7 -> 12 -> 85 -> 87 -> 8 -> 34 -> 20 -> 8 -> 87 -> 85 -> 12 -> 21 -> 49 -> 91 -> 49 -> 68 -> 49 -> 21 -> 50 ->
91 -> 68 -> 14 -> 36 -> 55 -> 97 -> 69 -> 75 -> 78 -> 75 -> 55 -> 97 -> 69 -> 23 -> 78 -> 37 -> 11 -> 3 -> 11 -> 1 -> 49
-> 3 -> 14 -> 36 -> 34 -> 36 -> 11 -> 1 -> 3 -> 68 -> 37 -> 78 -> 23 -> 75 -> 38 -> 75 -> 55 -> 52 -> 20 -> 73 -> 88 ->
32 -> 90 -> 88 -> 32 -> 35 -> 46 -> 71 -> 0 -> 93 -> 0, Gesamtlaenge: 2621481.0
Tag 2: 0 -> 4 -> 98 -> 45 -> 10 -> 98 -> 32 -> 81 -> 88 -> 90 -> 32 -> 46 -> 71 -> 0, Gesamtlaenge: 116670.0
Tag 3: 0, Gesamtlaenge: 0
Tag 4: 0 -> 44 -> 59 -> 62 -> 6 -> 26 -> 57 -> 79 -> 57 -> 92 -> 79 -> 92 -> 57 -> 26 -> 6 -> 62 -> 59 -> 24 -> 31 -> 70
-> 28 -> 70 -> 31 -> 24 -> 59 -> 44 -> 0, Gesamtlaenge: 345102.0
Tag 5: 0 -> 58 -> 45 -> 41 -> 5 -> 77 -> 18 -> 23 -> 97 -> 23 -> 38 -> 97 -> 38 -> 69 -> 78 -> 55 -> 52 -> 73 -> 88 -> 35
-> 93 -> 0, Gesamtlaenge: 240431.0
```

`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
```

Keine Ausgabe. (Iteration dauert zu lange)

`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
```

Keine Ausgabe. (Iteration dauert zu lange)

Wie man in den Beispielen 5-8 sieht, ist dieser Algorithmus selbst mit einem hohen dropout-Wert noch sehr langsam, da in jedem Iterationsschritt `(k-1)*(|Cmax|)` mit Cmax als dem längsten Pfad in C0 bis Ck Möglichkeiten berechnet werden müssen.

`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 -> 1 -> 2 -> 0, Gesamtlaenge: 51.7
Tag 2: 0, Gesamtlaenge: 0
Tag 3: 0, Gesamtlaenge: 0
Tag 4: 0, Gesamtlaenge: 0
Tag 5: 0, Gesamtlaenge: 0
Maximale Lange einer Tagestour: 51.7
```

Quellcode

utility.py

```
from typing import Any, Callable, List, Dict, Hashable


class TabuList:
    tabu: Dict[Hashable, int]
    offset: int
    default_tenure: int
    cleanup_freq: int

    def __init__(self, default_tenure: int, cleanup_freq: int = 20):
        self.tabu = {}
        self.offset = 0
        self.default_tenure = default_tenure
        self.cleanup_freq = cleanup_freq

    def _cleanup(self):
        to_delete = []
        for k, v in self.tabu.items():
            if v+self.offset <= 0:
                to_delete.append(k)
        for k in to_delete:
            del self.tabu[k]

    def add(self, item: Hashable, tenure: int = None):
        self.tabu[item] = (tenure or self.default_tenure)-self.offset

    def get(self, item: Hashable) -> int:
        if item in self.tabu:
            val = self.tabu[item]+self.offset
            return 0 if val < 0 else val
        return 0

    def tick(self):
        self.offset -= 1
        if self.offset % self.cleanup_freq == 0:
            self._cleanup()


def remove_by_exp(exp: Callable[[Any], bool], lst: List):
    for i in lst:
        try:
            if exp(i):
                lst.remove(i)
                break
        except Exception:
            pass
```

program.py

```
from collections import Counter
from os.path import dirname, join
from typing import (FrozenSet, Iterable, List, Mapping, Set,
                    Tuple)

from tabu_optimization import MMKCPP_TEE_TabuSearch
from utility import remove_by_exp


class CityGraph:
```

```python
    """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.

        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(
            list(range(n)),
            [(int(v), int(u), float(length)) for v, u, length in
             [line.split() for line in lines[1: m + 1]]])

    def __init__(self, vertices: List[int], edges: List[Tuple[int, int, float]]):
        self.vertices = {v: {} for v in vertices}
        self.edgeset = set()

        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 w_tour(self, tour: Tuple[int, ...]) -> float:
        return sum(self.vertices[tour[i]][tour[i+1]] for i in range(len(tour)-1))

    def is_connected(self) -> bool:
        unseen = set(self.vertices.keys())
        q = deque((0,))
        while q:
            current = q.popleft()
            if current not in unseen:
                continue
            unseen.remove(current)
            for next_ in self.vertices[current]:
                q.append(next_)
        return not unseen

    def get_paths(self, days: int = 5) -> List[Tuple[float, Tuple[int, ...]]]:
        return map(lambda x: (self.w_tour(x), x), MMKCPP_TEE_TabuSearch(self.vertices, days, 100, 600))


# repl
while True:
    pth = join(dirname(__file__),
                    f'beispieldaten/muellabfuhr{input("Bitte die Nummer des Beispiels eingeben [0-9]: ")}.txt')
    cg = CityGraph._from_bwinf_file(pth)
    n_days = int(input('Für wieviele Tage soll geplant werden? (5): ') or 5)
    maxlen = 0
    iterable = zip(range(1, n_days+1), cg.get_paths(n_days))
    for i, (len_, p) in iterable:
        print(f'Tag {i}: {" -> ".join(map(str, p))}, Gesamtlaenge: {len_}')
        maxlen = max(maxlen, len_)
    print(f'Maximale Lange einer Tagestour: {maxlen}')
```

tabu_optimization.py

```python
from collections import Counter, deque
from functools import reduce
from operator import add
from random import random
from time import time
from typing import Dict, List, Tuple, Iterable, Callable

from utility import TabuList

def MMKCPP_TEE_TabuSearch(G: Dict[int, Dict[int, float]], k: int = 5, maxNOfItsWithoutImprovement: int = 100,
                          maxRunningTime: float = None, dropout: float = 0.5, dropout_fn: Callable = lambda x: x**1.2,
                          tabuTenure: int = 20) -> List[Tuple[int, ...]]:
    """
    Generate a starting path and perform a meta-heuristic optimisation.

    Parameters
    ----------
    G : Dict[int, Dict[int, float]]
        The undirected, non-windy weighted graph to operate on.
    k : int, default=5
        Number of Vehicles.
    maxNOfItsWithoutImprovement : int, default=100
        Maximum number of iterations without improvement to stop early.
    maxRunningTime : float, optional
        The maximum running time to stop early (seconds).
    tabuTenure : int, default=20
        The number of iterations to 'tabu' a neighbor.

    Returns
    -------
    List[List[int]]
        An optimized List of k paths

    """
    # precompute shortest paths O(|V|²)
    dijkstra = {k: {} for k in G}  # shallowcopy doesnt work
    for start in dijkstra:
        q = deque(((0, start, []),))
        while q:
            length, current, currentpath = q.popleft()
            if current in dijkstra[start]: continue
            dijkstra[start][current] = (length, tuple(currentpath[1:]))
            for next_, weight in G[current].items():
                q.append((length+weight, next_, currentpath+[current]))

    def edges(tour: Tuple[int, ...]) -> Iterable[set]: #
        return (set(tour[i:][:2]) for i in range(len(tour)-1))

    def w_tour(tour: Tuple[int, ...]) -> float:
        return sum(G[tour[i]][tour[i+1]] for i in range(len(tour)-1))

    def w_max_tours(tours: Iterable[Tuple[int, ...]]) -> float:
        return max(w_tour(tour) for tour in tours)

    def edgecount_tour(tour: Tuple[int, ...]) -> Counter: #
        return Counter(frozenset(x) for x in edges(tour))

    def edgecount_tours(tours: List[Tuple[int, ...]]) -> Counter: #
        return reduce(add, (edgecount_tour(tour) for tour in tours))

    def MergeWalkWithTour(tour: Tuple[int, ...], walk: Tuple[int, ...]) -> Tuple[int, ...]: #
        # remove edges from walk that are already in tour
        if len(walk) == 1:
            return tour

        walk = list(walk)

        tour_edges = edges(tour)
        if not tour_edges:
            return ((0,) if walk[0] != 0 else ())+dijkstra[0][walk[0]][1]+tuple(walk)+dijkstra[walk[-1]][0][1]+((0,) if
walk[-1] != 0 else ())

        while tour_edges:
```

```
            if frozenset((walk[0], walk[1])) in tour_edges:
                del walk[0]
                if len(walk) == 1:
                    return tour
            else:
                break

        while tour_edges:
            if frozenset((walk[-1], walk[-2])) in tour_edges:
                del walk[-1]
                if len(walk) == 1:
                    return tour
            else:
                break

        walk = tuple(walk)

        # find node `t` closest to `u` and `v`, the end nodes of `walk`
        min_idx = None
        min_distance = 999999
        min_sp_v = min_sp_u = None

        for i, node in enumerate(tour):
            sp_u = dijkstra[node][walk[0]]
            sp_v = dijkstra[walk[-1]][node]
            if sp_u[0]+sp_v[0] < min_distance:
                min_distance = sp_v[0]+sp_u[0]
                min_idx = i
                min_sp_u = sp_u[1]
                min_sp_v = sp_v[1]

        # splice
        return tour[:min_idx+(1 if tour[min_idx] != walk[0] else 0)]+min_sp_u+walk+min_sp_v+tour[min_idx+(1 if
tour[min_idx] == walk[-1] else 0):]


    # basically shortenPath
    def SeparateWalkFromTour(tour: Tuple[int, ...], walk: Tuple[int, ...]) -> Tuple[int, ...]:
        # assuming walk is a subsegment of tour
        u, v, = walk[0], walk[-1]

        # better lr ri finding
        for i in range(len(tour)-2):
            if walk == tour[i:][:3]:
                li = i
                ri = i+2
                break

        return tour[:li+(1 if u != v else 0)]+dijkstra[u][v][1]+tour[ri:]

    def ReorderToClosedWalk(edgeset: List[set]) -> Tuple[int, ...]:
        newtour = [0]  # depot node

        while edgeset:
            stop = True
            for edge in edgeset:
                if newtour[-1] in edge:
                    edge.remove(newtour[-1])
                    newtour.append(edge.pop())
                    edgeset.remove(edge)
                    stop = False
            if stop: break

        while edgeset:  # find walks and append them to the main path
            walk = list(edgeset.pop())
            while True:
                stop = True
                for edge in edgeset:
                    if newtour[-1] in edge:
                        edge.remove(walk[-1])
                        walk.append(edge.pop())
                        edgeset.remove(edge)
                        stop = False
                if stop: break
            newtour = list(MergeWalkWithTour(tuple(newtour), tuple(walk)))
```

```
    return tuple(newtour)

def RemoveEvenRedundantEdges(tour: Tuple[int, ...], tours: List[Tuple[int, ...]]) -> Tuple[int, ...]:
    edgeset = list(edges(tour))
    for edge in map(frozenset, edgeset):
        ects = edgecount_tours(tours)[edge]
        ect = edgecount_tour(tour)[edge]
        if ects > ect and ect % 2 == 0:
            # check if tour remains connected to node 0 when removing edge 2x
            nodes = set((0,))
            remaining = set(map(frozenset, edges(tour)))
            remaining.discard(edge)
            if ect > 2:
                remaining.clear()  # skip to else if no connection is in danger
            while remaining:
                stop = True
                to_remove = None
                for edge_ in remaining:
                    if nodes.intersection(edge_):
                        nodes.update(edge_)
                        to_remove = edge_
                        stop = False
                        break
                if to_remove:
                    remaining.remove(to_remove)
                if stop:
                    break
            else:
                # remove 2 edges
                edgeset.remove(edge)
                edgeset.remove(edge)
    return ReorderToClosedWalk(edgeset)

# first make a starting solution
# all edges in graph + dijkstra between odd connections
edges_ = list(map(set, list(set(frozenset((start, end)) for start in G for end in G[start]))))
odd = [k for k, v in G.items() if len(v) % 2]
for _ in range(0, len(odd), 2):
    odd1 = odd.pop()
    odd2 = odd.pop()
    edges_ += list(map(set, edges((odd1,)+dijkstra[odd1][odd2][1]+(odd2,))))

singlePath = ReorderToClosedWalk(edges_)

bestSolution = [singlePath]+[(0,)]*(k-1)
currentSolution = bestSolution

bestSolutionValue = w_max_tours(bestSolution)
currentSolutionValue = w_max_tours(bestSolution)

nOfItsWithoutImprovement = 0

tabuList = TabuList(tabuTenure)
allEdgesCnt = len(set.union(set(map(frozenset, edges(bestSolution)))))

if maxRunningTime:
    startTime = time()

while (nOfItsWithoutImprovement < maxNOfItsWithoutImprovement and not
       (maxRunningTime and time() > startTime + maxRunningTime)):
    nOfItsWithoutImprovement += 1
    tabuList.tick()

    neighborhood: List[Tuple[Tuple[int]]] = []

    # compute neighborhood
    current_max_tour = max(currentSolution, key=w_tour)
    current_max_tour_idx = currentSolution.index(current_max_tour)

    for i in range(len(current_max_tour)-2):
        semilocal_tours = currentSolution.copy()
        walk = current_max_tour[i:i+3]  # 3 nodes, 2 edges
        semilocal_tours[current_max_tour_idx] = SeparateWalkFromTour(current_max_tour, walk)
        semilocal_tours[current_max_tour_idx] = RemoveEvenRedundantEdges(semilocal_tours[current_max_tour_idx],
```

```
semilocal_tours)

            for other_tour_idx in range(k):
                if other_tour_idx == current_max_tour_idx or random() <= dropout:
                    continue
                local_tours = semilocal_tours.copy()
                other_tour = local_tours[other_tour_idx]

                local_tours[other_tour_idx] = MergeWalkWithTour(other_tour, walk)
                local_tours[other_tour_idx] = RemoveEvenRedundantEdges(local_tours[other_tour_idx], local_tours)

                neighborhood.append(tuple(local_tours))


        # filter tabu, reduce max length
        try:
            currentSolution = min(filter(lambda x: not tabuList.get(x), neighborhood), key=lambda x: (w_max_tours(x),
random()))
        except ValueError:  # no non-tabu neighbors, were done
            return bestSolution
        tabuList.add(currentSolution)
        currentSolution = list(currentSolution)
        currentSolutionValue = w_max_tours(currentSolution)

        if currentSolutionValue < bestSolutionValue:
            print('.', end='')
            nOfItsWithoutImprovement = 0
            bestSolutionValue = currentSolutionValue
            bestSolution = currentSolution

        dropout = dropout_fn(dropout)

    print()
    return bestSolution
```