Fahrradwerkstatt

❔ A4 👥 00122 👴 Leonhard Masche 🗓️ 9.09.2022

Lösungsidee

Die Idee zur Lösung ist, den Zeitpunkt der Aufgabe des Auftrags und dessen Dauer in einer Liste zu speichern. Nun werden die Aufträge mit den gegebenen Methoden (nach Zeitpunkt oder Dauer) bearbeitet und die Ergebnisse ausgegeben.

Umsetzung

Das Programm ist in Python umgesetzt und mit einer Umgebung ab der Version `3.6` ausführbar. Das Programm befindet sich in der Datei `program.py` in diesem Ordner.

Zuerst werden die Aufträge aus dem Beispiel in eine Liste geladen. Die Aufträge werden dabei als Tuple gespeichert.

Nun werden die Aufträge mit den gegebenen Methoden bearbeitet. Hier gibt zusätzlich zu den vorgeschlagenen Methoden `by_submit` (Die Aufträge werden in der Reihenfolge ihrer Aufgabe bearbeitet) und `by_duration` (Unter den verfügbaren Aufträgen wird immer der Kürzeste ausgewählt) auch noch `by_duration_resumable`, die im Folgenden beschrieben wird. Die Ergebnisse werden in einem DataFrame aus der Bibliothek `pandas` gespeichert, um eine formatierte Ausgabe zu erleichtern.

Zum Schluss werden die Ergebnisse formatiert ausgegeben.

Verbesserungen

Wie die Simulation zeigt, wird durch die Auswahl des Auftrags mit der kürzesten Dauer die Durchschnittszeit der Bearbeitung der Aufträge deutlich reduziert. Dadurch verschlechtert sich aber gleichzeitig die maximale Wartezeit, da längere Aufträge immer weiter 'nach hinten' in der Warteschlange geschoben werden. Kunden mit längeren Aufträgen werden daher unzufriedener sein, da ihre Aufträge zugunsten von kürzeren Aufträgen immer wieder "unfairerweise" weiter nach Hinten in der Warteschlange verschoben werden.

Achtet man jedoch nur auf die durchschnittliche Bearbeitungszeit, lässt sich eine noch bessere Bearbeitungsmethode für Warteschlangen des Typs `G/G/1` finden:

`by_duration_resumable`

Es wird angenommen, dass jeder Auftrag niedergelegt, ein kürzerer Auftrag aufgenommen werden, und nach Fertigstellung der vorherige Auftrag wieder aufgenommen werden kann. So kann immer der kürzeste Auftrag bearbeitet werden, und nicht nur wenn ein neuer Auftrag ausgewählt wird. Dies geht mit einer deutlichen Verbesserung der durchschnittlichen Wartezeit einher.

Dies ist die optimale Art eine `G/G/1` Warteschlange zu bearbeiten, da zu jedem Zeitpunkt der kürzeste Auftrag bearbeitet wird und die Zeit bis zur Fertigstellung eines Auftrags immer minimal ist. Diese Verbesserung ist auch deutlich in der Simulation zu erkennen.

Beispiele

Hier wird das Programm auf die fünf Beispiele von der Website angewendet:

`fahrradwerkstatt0.txt`

```
Aufträge simuliert in 331.3ms:

┌──────────────────────────────────┬───────────┬─────────────┬──────────────────────┐
│                                  │ by_submit │ by_duration │ by_duration_resumable │
├──────────────────────────────────┼───────────┼─────────────┼──────────────────────┤
│ Maximale Wartezeit (min)         │     68771 │      188734 │               344634 │
├──────────────────────────────────┼───────────┼─────────────┼──────────────────────┤
│ Durchschnittliche Wartezeit (min) │   32753.5 │     16981.3 │              13928.1 │
└──────────────────────────────────┴───────────┴─────────────┴──────────────────────┘
```

`fahrradwerkstatt1.txt`

```
Aufträge simuliert in 407.9ms:

┌──────────────────────────────────┬───────────┬─────────────┬──────────────────────┐
│                                  │ by_submit │ by_duration │ by_duration_resumable │
├──────────────────────────────────┼───────────┼─────────────┼──────────────────────┤
│ Maximale Wartezeit (min)         │    128321 │      433563 │               433563 │
├──────────────────────────────────┼───────────┼─────────────┼──────────────────────┤
│ Durchschnittliche Wartezeit (min) │   63535.7 │     11883.9 │              11106.5 │
└──────────────────────────────────┴───────────┴─────────────┴──────────────────────┘
```

`fahrradwerkstatt2.txt`

```
Aufträge simuliert in 291.5ms:

┌──────────────────────────────────┬───────────┬─────────────┬──────────────────────┐
│                                  │ by_submit │ by_duration │ by_duration_resumable │
├──────────────────────────────────┼───────────┼─────────────┼──────────────────────┤
│ Maximale Wartezeit (min)         │    110973 │      327087 │               331392 │
├──────────────────────────────────┼───────────┼─────────────┼──────────────────────┤
│ Durchschnittliche Wartezeit (min) │   51194.5 │     14805.7 │              10969.2 │
```

fahrradwerkstatt3.txt

	by_submit	by_duration	by_duration_resumable
Maximale Wartezeit (min)	60821	382016	382016
Durchschnittliche Wartezeit (min)	30028.9	17242.8	16504.1

fahrradwerkstatt4.txt

	by_submit	by_duration	by_duration_resumable
Maximale Wartezeit (min)	167059	363155	363155
Durchschnittliche Wartezeit (min)	74427.5	42200.9	40512.8

Quellcode

program.py

```
from copy import deepcopy
from math import floor
from os import path
from time import time
from typing import Any, Callable, Iterable, List, Tuple
from pandas import DataFrame


start_time = 9*60
end_time = 17*60


def r_path(path_: str) -> str:
    """Return the absolute path to a file relative to the current file.

    Parameters
```

```
    ----------
    path_ : str
        The path to the file relative to the current file.

    Returns
    -------
    str
        The absolute path to the file.
    """
    return path.join(
        path.dirname(path.abspath(__file__)),
        path_
    )


class TOD:
    """Time Of Day. Automatically jumps over the gaps in working time."""

    start_time: int
    end_time: int
    value: int

    def __init__(self, start_time: int, end_time: int):
        """Initialize the TOD.

        Parameters
        ----------
        start_time : int
            The time the working day starts in minutes from 00:00.
        end_time : int
            The time the working day ends in minutes from 00:00.
        """
        self.start_time = start_time
        self.end_time = end_time
        self.value = start_time

    def get(self) -> int:
        """Return the current time.

        Returns
        -------
        int
            The current time in minutes.
        """
        return self.value

    def last_end(self) -> int:
        """Return the time the last timestep was finished.

        This only applies if an order is completed.

        Returns
        -------
        int
            The time the last timestep was finished at.
        """
```

```
        if self.value % (24*60) == self.start_time:
            days = floor(self.value // (24*60))
            return (days-1)*24*60+self.end_time
        return self.value

    def add(self, minutes: int):
        """Add minutes to the current time while accounting for gaps in
        working time.
        """
        days = minutes // (self.end_time - self.start_time)
        minutes_ = minutes % (self.end_time - self.start_time)
        self.set(self.value + days*24*60 + minutes_)

    def set(self, time: int):
        """Set the current time while accounting for gaps in working time."""
        days = floor(time / (24*60))
        minutes = time % (24*60)
        if minutes < self.start_time:
            self.value = days*24*60+self.start_time
        elif minutes >= self.end_time:
            self.value = (days+1)*24*60+self.start_time
        else:
            self.value = time


class Sim:
    """Simulate the processing of orders."""

    tod: TOD
    orders: Iterable[Tuple[int, int]]
    processing: List[List[int]]
    key: Callable[[Any], float]
    resumable: bool
    on_done: Callable[[int, int], None]

    def __init__(self, queue: Iterable[Tuple[int, int]], start_time: int, end_time: int):
        self.tod = TOD(start_time, end_time)    # create TOD object
        self.orders = deepcopy(queue)            # create a local copy of the orders queue
        self.orders.sort(key=lambda x: x[0])     # sort by submit
        self.processing = []
        self.key = lambda x: x[0]               # default sorting function for processing
orders
        self.resumable = False                  # default to not resumable

    def __bool__(self) -> bool:
        # return true if there are still orders to process
        return bool(self.orders) or bool(self.processing)

    def set_key(self, key: Callable[[Any], float]):
        """Set the sorting key for which orders to prioritize.

        Parameters
        ----------
        key : Callable[[Any], float]
            A function that takes an order and returns a number that can be compared.
        """
```

```
        self.key = key

    def set_resumable(self, resumable: bool):
        """Set whether an order has to be completed first before another can be picked up.

        Parameters
        ----------
        resumable : bool
            Whether an order has to be completed first before another can be picked up.
        """
        self.resumable = resumable

    def set_callback(self, cb: Callable[[int, int], None]):
        """Set a callback for when an order is completed.

        Parameters
        ----------
        cb : Callable[[int, int], None]
            The callback to call withthe submit time and the end time of the
            order when it is finished.
        """
        self.on_done = cb

    def _update_processing(self):
        # move orders from orders to processing
        if len(self.processing) == 0 and self.orders:
            self.tod.set(self.orders[0][0])

        while self.orders and self.orders[0][0] <= self.tod.get():
            self.processing.append(list(self.orders.pop(0)))

    def run_until_end(self):
        """Run the simulation until all orders are processed.

        on_done is called for each order that is completed.
        """
        previous_order_done = True
        while self:
            self._update_processing()
            if self.resumable or previous_order_done:
                previous_order_done = False
                self.processing.sort(key=self.key)
            self.tod.add(1)
            self.processing[0][1] -= 1
            if self.processing[0][1] == 0:
                self.on_done(self.processing[0][0], self.tod.last_end())
                self.processing.pop(0)
                previous_order_done = True


def by_submit(orders: Iterable[Tuple[int, int]]) -> Tuple[int, float]:
    """Process orders in the order they were submitted and return the
    simulated wait times.

    Returns
    -------
```

```
    Tuple[int, float]
        [average wait time, maximum wait time]
    """
    avg_wait: float = 0
    n_avg_wait: int = 0
    max_wait: int = 0

    def on_done(submit: int, done: int):
        nonlocal avg_wait, n_avg_wait, max_wait
        wait = done - submit
        avg_wait = (n_avg_wait * avg_wait + wait) / (n_avg_wait + 1)
        n_avg_wait += 1
        max_wait = max(max_wait, wait)

    sim = Sim(orders, start_time, end_time)
    sim.set_key(lambda x: x[0])
    sim.set_resumable(False)
    sim.set_callback(on_done)

    sim.run_until_end()

    return max_wait, avg_wait


def by_duration(orders: Iterable[Tuple[int, int]]) -> Tuple[int, float]:
    """Process orders sorted by their duration and return the
    simulated wait times.

    Returns
    -------
    Tuple[int, float]
        [average wait time, maximum wait time]
    """
    avg_wait: float = 0
    n_avg_wait: int = 0
    max_wait: int = 0

    def on_done(submit: int, done: int):
        nonlocal avg_wait, n_avg_wait, max_wait
        wait = done - submit
        avg_wait = (n_avg_wait * avg_wait + wait) / (n_avg_wait + 1)
        n_avg_wait += 1
        max_wait = max(max_wait, wait)

    sim = Sim(orders, start_time, end_time)
    sim.set_key(lambda x: x[1])
    sim.set_resumable(False)
    sim.set_callback(on_done)

    sim.run_until_end()

    return max_wait, avg_wait


def by_duration_resumable(orders: Iterable[Tuple[int, int]]) -> Tuple[int, float]:
    """Process orders in the order they were submitted (resumable)
```

```
    and return the simulated wait times.

    Returns
    -------
    Tuple[int, float]
        [average wait time, maximum wait time]
    """
    avg_wait: float = 0
    n_avg_wait: int = 0
    max_wait: int = 0

    def on_done(submit: int, done: int):
        nonlocal avg_wait, n_avg_wait, max_wait
        wait = done - submit
        avg_wait = (n_avg_wait * avg_wait + wait) / (n_avg_wait + 1)
        n_avg_wait += 1
        max_wait = max(max_wait, wait)

    sim = Sim(orders, start_time, end_time)
    sim.set_key(lambda x: x[1])
    sim.set_resumable(True)
    sim.set_callback(on_done)

    sim.run_until_end()

    return max_wait, avg_wait


# main loop
def main():
    print()
    example_n = int(input('Bitte die Nummer des Beispiels eingeben [0; 4]: '))

    print()

    start_time = time()  # time measurement

    queue: List[Tuple[int]] = []  # [[submit, length]]
    with open(r_path(f'beispieldaten/fahrradwerkstatt{example_n}.txt'), 'r', encoding='utf-8')
as f:
        for a, b in map(lambda x: tuple(map(int, x.split(''))),
                        filter(lambda x: x != '', f.read().split('\n'))):
            queue.append((a, b))

    processors = [by_submit, by_duration, by_duration_resumable]

    data = {processor.__name__: processor(queue)
            for processor in processors}   # apply all processors

    print(f'Aufträge simuliert in {format((time()-start_time)*1000, ".1f")}ms:')
    print()

    df = DataFrame(data, index=['Maximale Wartezeit (min)',
                    'Durchschnittliche Wartezeit (min)'])
    print(df.to_markdown(tablefmt='rounded_grid'))
```

```
# programmschleife
print('Fahrradwerkstatt')
print('(Drücke ^C um das Programm zu beenden)')

while True:
    try:
        main()
    except Exception as e:
        print(e)
    except KeyboardInterrupt:
        print('\n')
        exit()
```