Mateusz Woźniak 260292 03.04.2023

Sztuczna inteligencja i inżynieria wiedzy – Raport lista 1

Cel:

Celem ćwiczenia jest praktyczne zapoznanie się z problemami optymalizacyjnymi oraz praktyczne przećwiczenie omawianych na wykładzie metod rozwiązywania pewnej podklasy tych problemów. Po wykonaniu listy, student powinien wiedzieć czym jest problem optymalizacyjny, jakie trudności mogą się wiązać z uzyskaniem dokładnego rozwiązania przedstawionego problemu oraz jak poradzić sobie z rozwiązaniem problemu przy ograniczonych zasobach (np. moce obliczeniowe, czas). W szczególności znane powinny być różnice między aproksymacją a heurystyką oraz przykłady podejścia heurystycznego do problemu przeszukania, oraz znajdowania ścieżek.

Dane do wykonania ćwiczenia:

Plik *connection_graph.csv*:

connection_graph.csv

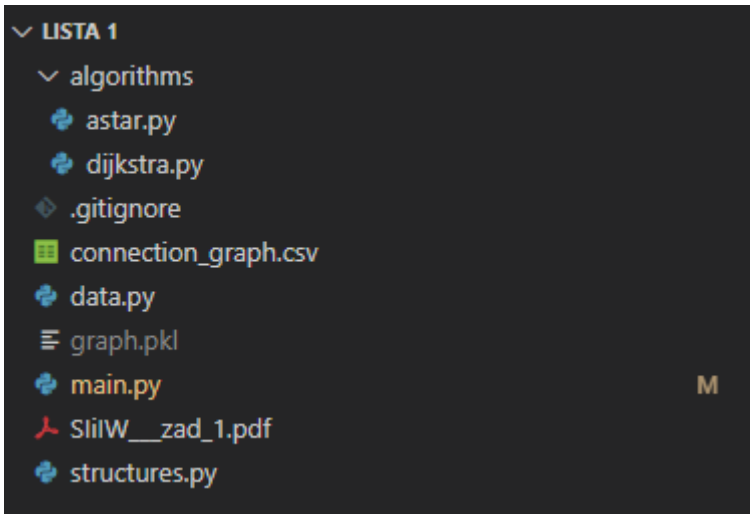
```
,company,line,departure_time,arrival_time,start_stop,end_stop,start_stop_lat,start_stop_lon,end_stop_lat,end_stop_lon
0,MPK Autobusy,A,20:00:00,20:01:00,KRZYKI,Sowia,51.07488366,17.00656861,51.07379262,17.00184494
1,MPK Autobusy,A,20:01:00,20:02:00,Sowia,Chłodna,51.07379262,17.00184494,51.07512234,16.99667086
2,MPK Autobusy,A,20:02:00,20:03:00,Chłodna,Wawrzyniaka,51.07512234,16.99667086,51.078074,16.9982022
3,MPK Autobusy,A,20:03:00,20:05:00,Wawrzyniaka,Rymarska,51.078074,16.9982022,51.0793232,16.99125798
4,MPK Autobusy,A,20:05:00,20:06:00,Rymarska,RACŁAWICKA,51.0793232,16.99125798,51.077395,16.983938
5,MPK Autobusy,A,20:06:00,20:07:00,RACŁAWICKA,Bukowskiego,51.077395,16.983938,51.07670719,16.97736107
6,MPK Autobusy,A,20:07:00,20:08:00,Bukowskiego,Stanki,51.07670719,16.97736107,51.07632585,16.97206692
7,MPK Autobusy,A,20:08:00,20:09:00,Stanki,Kadłubka,51.07632585,16.97206692,51.07647653,16.96866058
8,MPK Autobusy,A,20:09:00,20:10:00,Kadłubka,Wiejska,51.07647653,16.96866058,51.07763602,16.96391602
9,MPK Autobusy,A,20:10:00,20:11:00,Wiejska,Solskiego,51.07763602,16.96391602,51.08179036,16.96866979
10,MPK Autobusy,A,20:11:00,20:13:00,Solskiego,GRABISZYŃSKA (Cmentarz),51.08179036,16.96866979,51.0896217,16.97666221
11,MPK Autobusy,A,20:13:00,20:15:00,GRABISZYŃSKA (Cmentarz),FAT,51.0896217,16.97666221,51.09375561,16.98037615
12,MPK Autobusy,A,20:15:00,20:17:00,FAT,Aleja Pracy,51.09375561,16.98037615,51.09225862,16.98641641
13,MPK Autobusy,A,20:17:00,20:18:00,Aleja Pracy,Inżynierska,51.09225862,16.98641641,51.09454012,16.99196391
14,MPK Autobusy,A,20:18:00,20:20:00,Inżynierska,Krucza (Mielecka),51.09454012,16.99196391,51.09325606,17.00249247
15,MPK Autobusy,A,20:20:00,20:21:00,Krucza (Mielecka),Krucza,51.09325606,17.00249247,51.09342283,17.0083676
16,MPK Autobusy,A,20:21:00,20:23:00,Krucza,Pl. Hirszfelda,51.09342283,17.0083676,51.09382147,17.01795905
17,MPK Autobusy,A,20:23:00,20:28:00,Pl. Hirszfelda,Arkady (Capitol),51.09382147,17.01795905,51.10203906,17.02983653
18,MPK Autobusy,A,20:28:00,20:31:00,Arkady (Capitol),Dworzec Główny (Dworcowa),51.10203906,17.02983653,51.10114919,17.04001736
```

Którego kolumny oznaczają kolejno:

a) id przejazdu
b) company – przewoźnik
c) line – nazwa linii
d) departure_time – czas odjazdu (hh:mm:ss)
e) arrival_time – czas przyjazdu (hh:mm:ss)
f) start_stop – nazwa przystanku początkowego
g) end_stop – nazwa przystanku końcowego
h) start_stop_lat – szerokość geograficzna przystanku początkowego
i) end_stop_lat – szerokość geograficzna przystanku końcowego
j) start_stop_lon – długość geograficzna przystanku początkowego
k) end_stop_lon – długość geograficzna przystanku końcowego

Informacje na temat projektu:

a) język programowania – Python
b) implementowane algorytmy – algorytm Dijkstry, algorytm A*
c) struktura plikowa:



Dane są serializowane do pliku *graph.pkl*, aby uniknąć ich wielokrotnego przetwarzania. Graf jest reprezentowany za pomocą listy węzłów (przystanków), w których zawarte są listy sąsiedztw (listy odjazdów).

data.py > fetchNodes

```python
import pandas as pd
from collections import defaultdict
from structures import Stop
import pickle
import time

df = pd.read_csv('connection_graph.csv', index_col=0)
FILENAME = "graph.pkl"

def fetchNodes():
    stops_dict = defaultdict(set)
    print("Fetching nodes...")
    for _, elem in df.iterrows():
        stops_dict[elem['start_stop']].add(tuple(elem[6:8]))
        stops_dict[elem['end_stop']].add(tuple(elem[8:]))

    return list(sorted([Stop(stop[0], stop[1]) for stop in stops_dict.items()], key=lambda x: x.name))

def fetchEdges(graph):
    print("Fetching edges...")
    for _, elem in df.sort_values(by='start_stop').iterrows():
        for stop in graph:
            if stop.name==elem['start_stop'] and stop.name!=elem['end_stop']:
                stop.addDeparture(elem)
                break

def saveGraph(graph):
    print("Saving...")
    with open(FILENAME, "wb") as file:
        file.write(pickle.dumps(graph))

def readGraph():
    with open(FILENAME, "rb") as file:
        return pickle.loads(file.read())

def fullLoad():
    a = time.time()
    graph = fetchNodes()
    fetchEdges(graph)
    saveGraph(graph)
    print(f"Done in {time.time()-a}s!")
```

Algorytm Dijkstry

Algorytm Dijkstry to popularny algorytm służący do znajdowania najkrótszej ścieżki między wierzchołkami w grafie ważonym. Na podstawie iteracji, na bieżąco poprawiane są najkrótsze ścieżki od początku drogi do pozostałych wierzchołków.
W implementacji tego algorytmu skorzystałem z kolejki priorytetowej, a po ukończeniu iteracji, droga została odtworzone idąc wstecz wzdłuż przystanków na podstawie słownika zawierającego min. poprzedni przystanek.

```python
from structures import PriorityQueue, Stop, findStopOfName, timeToTotal, findDepartureBetween, preetifyResult
import time

def dijkstra(graph, start, end, departure_time):
    """
    Implementation of a Dijkstra algorithm
    -
    graph - graph used to browse for an optimal route\n
    start - starting stop name\n
    end - end goal\n
    departure_time - the earliest time to hop onto a bus/tram
    """
    computing_time_start = time.time()

    start_node = findStopOfName(graph, start)
    stop_node = findStopOfName(graph, end)
    time_total = timeToTotal(departure_time)

    q = PriorityQueue()
    q.put(start_node, 0)
    visited = set()

    tracker = {stop: (Stop, float('inf'), int) for stop in graph}
    # dict storing current best paths to a node, where:
    # first elem - path to node
    # second elem - path length
    tracker[start_node] = (None, 0, time_total)
    while not q.empty():
        current_stop = q.get()

        if current_stop in visited:
            continue
        else:
            visited.add(current_stop)

        if current_stop == stop_node:
            # path reconstruction
            stopA = tracker[current_stop][0]
            stopB = current_stop
            departures = []
            while stopA != start_node:
                departures.append(findDepartureBetween(stopA, stopB, tracker[stopB][2]))
                stopA, stopB = tracker[stopA][0], stopA
            departures.append(findDepartureBetween(stopA, stopB, tracker[stopB][2]))
            departures.reverse()
            preetifyResult(departures)
            print(f"Czas wykonywania obliczeń - {time.time()-computing_time_start}s")
            return

        collection = set(filter(lambda x: (x.departure_time>=tracker[current_stop][2]), current_stop.departures))

        for departure in collection:
            # all departures that take place after current_stop arrival time
            destination = findStopOfName(graph, departure.destination)
            if destination.name in [current_stop.name, tracker[current_stop][0]]:
                continue
            time_cost = departure.timeCriteria(time_total)

            if not tracker[destination][0] or time_cost < tracker[destination][1]:
                tracker[destination] = (current_stop, time_cost, departure.arrival_time)
                q.put(destination, time_cost)
    return
```

Algorytm A*

Algorytm ten rozszerza algorytm Dijkstry o heurystykę, która zwiększa efektywność przeszukiwania grafu. W wyżej wspomnianej kolejce priorytetowej, kolejka jest ustawiana na podstawie sumy kosztu oraz wartości heurystyki.
W zadaniu rozważane były 2 przypadki – kryterium przesiadkowe, a także kryterium czasowe, które kładły większy nacisk na odpowiednie wartości. W przypadku kryterium przesiadkowego aplikowano dodatkowe kary w wysokości 10 za przesiadkę.

```python
from structures import PriorityQueue, Stop, findStopOfName, timeToTotal, findDepartureBetween, preetifyResult
import time

def aStar(graph, start, end, departure_time, criteria):
    match criteria:
        case 't':
            asaTimeCriteria(graph,start,end,departure_time)
        case 'p':
            asaTransferCriteria(graph,start,end,departure_time)
        case _:
            print("Nie ma takiego kryterium")

def asaTimeCriteria(graph, start, end, departure_time):
    """
    Implementation of an A* Algorithm using time criteria
    -
    graph - graph used to browse for an optimal route\n
    start - starting stop name\n
    end - end goal\n
    departure_time - the earliest time to hop onto a bus/tram
    """
    computing_time_start = time.time()

    start_node = findStopOfName(graph, start)
    stop_node = findStopOfName(graph, end)
    time_total = timeToTotal(departure_time)

    q = PriorityQueue()
    q.put(start_node, 0)
    visited = set()

    tracker = {stop: (Stop, float('inf'), int) for stop in graph}
    # dict storing current best paths to a node, where:
    # first elem - path to node
    # second elem - path length
    tracker[start_node] = (None, 0, time_total)
```

```python
    while not q.empty():
        current_stop = q.get()

        if current_stop in visited:
            continue
        else:
            visited.add(current_stop)

        if current_stop == stop_node:
            # path reconstruction
            stopA = tracker[current_stop][0]
            stopB = current_stop
            departures = []
            while stopA != start_node:
                departures.append(findDepartureBetween(stopA, stopB, tracker[stopB][2]))
                stopA, stopB = tracker[stopA][0], stopA
            departures.append(findDepartureBetween(stopA, stopB, tracker[stopB][2]))
            departures.reverse()
            preetifyResult(departures)
            print(f"Czas wykonywania obliczeń - {time.time()-computing_time_start}s")
            return

        for departure in filter(lambda x: (x.departure_time>=tracker[current_stop][2]), current_stop.departures):
            comp = time.time()
            # all departures that take place after current_stop arrival time
            destination = findStopOfName(graph, departure.destination)
            if destination.name in [current_stop.name, tracker[current_stop][0]]:
                continue
            time_cost = departure.timeCriteria(time_total)

            if not tracker[destination][0] or time_cost < tracker[destination][1]:
                tracker[destination] = (current_stop, time_cost, departure.arrival_time)
                q.put(destination, time_cost + current_stop.getHeuristic(destination))
    return
```

```python
def asaTransferCriteria(graph, start, end, departure_time):
    """
    Implementation of an A* Algorithm using time criteria
    -
    graph - graph used to browse for an optimal route\n
    start - starting stop name\n
    end - end goal\n
    departure_time - the earliest time to hop onto a bus/tram
    """
    computing_time_start = time.time()

    start_node = findStopOfName(graph, start)
    stop_node = findStopOfName(graph, end)
    time_total = timeToTotal(departure_time)

    q = PriorityQueue()
    q.put(start_node, 0)
    visited = set()

    tracker = {stop: (Stop, float('inf'), int, str) for stop in graph}
    # dict storing current best paths to a node, where:
    # first elem - path to node
    # second elem - path length
    tracker[start_node] = (None, 0, time_total, None)
```

```python
    while not q.empty():
        current_stop = q.get()
        current_time = time_total

        if current_stop in visited:
            continue
        else:
            visited.add(current_stop)

        if current_stop == stop_node:
            # path reconstruction
            stopA = tracker[current_stop][0]
            stopB = current_stop
            departures = []
            while stopA != start_node:
                departures.append(findDepartureBetween(stopA, stopB, tracker[stopB][2]))
                stopA, stopB = tracker[stopA][0], stopA
            departures.append(findDepartureBetween(stopA, stopB, tracker[stopB][2]))
            departures.reverse()
            preetifyResult(departures)
            print(f"Czas wykonywania obliczeń - {time.time()-computing_time_start}s")
            return

        for departure in filter(lambda x: (x.departure_time>=tracker[current_stop][2]), current_stop.departures):
            # all departures that take place after current_stop arrival time
            destination = findStopOfName(graph, departure.destination)

            transfer = departure.line != tracker[current_stop][3]

            if destination.name in [current_stop.name, tracker[current_stop][0]]:
                continue
            time_cost = departure.transferCriteria(current_time, transfer)

            if not tracker[destination][0] or time_cost < tracker[destination][1]:
                tracker[destination] = (current_stop, time_cost, departure.arrival_time, departure.line)
                q.put(destination, time_cost + current_stop.getHeuristic(destination))
        current_time += tracker[current_stop][1]
    return
```

Heurystyka oraz kara związana z przesiadką:

```
def getHeuristic(self, end_stop):
    """
    Getter for heuristic value, using Manhattan Distance.\n
    Updates total f value.
    """
    MULTPLIER = 300
    this_x, this_y = self.posts[0]
    end_x, end_y = end_stop.posts[0]
    return MULTPLIER * (sqrt(abs(this_x-end_x) + abs(this_y-end_y)))
```

Heurystyka jest zawarta w klasie Stop

```
def timeCriteria(self, start_time):
    return self.arrival_time - start_time

def transferCriteria(self, start_time, transfered):
    TRANSFER_THRESHOLD = 10
    return self.arrival_time - start_time + (TRANSFER_THRESHOLD if transfered else 0)
```

Obie funkcje są zawarte w klasie Departure

Wypisanie wyników:

```
def preetifyResult(res):
    print("=======================================================")
    transfers = set()
    for departure in res:
        transfers.add(departure.line)
        print(f"{departure.start} -> {departure.destination}, linia {departure.line},\
              odjazd o {toReadableTime(departure.departure_time)}, przyjazd o {toReadableTime(departure.arrival_time)}")
    print(f"Docierasz do przystanku o {toReadableTime(res[-1].arrival_time)} po {res[-1].departure_time-res[0].departure_time} minutach.")
    print(f"Ilość przesiadek - {len(transfers)-1}")
    print("=======================================================")
```

Modyfikacja:

Usunąłem z przeglądania powtarzające się połączenia (odjazdy tych samych linii w tym samym kierunku, lecz o innych godzinach) pozostawiając jedynie najwcześniejsze z nich.

Przed:

```
    collection = set(filter(lambda x: (x.departure_time>=tracker[current_stop][2]), current_stop.departures))

    for departure in collection:
        # all departures that take place after current_stop arrival time
        destination = findStopOfName(graph, departure.destination)
        if destination.name in [current_stop.name, tracker[current_stop][0]]:
            continue
        time_cost = departure.timeCriteria(time_total)

        if not tracker[destination][0] or time_cost < tracker[destination][1]:
            tracker[destination] = (current_stop, time_cost, departure.arrival_time)
            q.put(destination, time_cost)
return
```

Czas wykonywania obliczeń - 9.529773235321045s

Po:

```
50
51          collection = sorted(filter(lambda x: (x.departure_time>=tracker[current_stop][2]), current_stop.departures), key=lambda x: x.departure_time)
52
53          checked_departures = set()
54
55          for departure in collection:
56              # all departures that take place after current_stop arrival time
57              x = (departure.line, departure.destination)
58              if x not in checked_departures:
59                  checked_departures.add(x)
60                  destination = findStopOfName(graph, departure.destination)
61                  if destination.name not in [current_stop.name, tracker[current_stop][0]]:
62                      time_cost = departure.timeCriteria(time_total)
63
64                      if not tracker[destination][0] or time_cost < tracker[destination][1]:
65                          tracker[destination] = (current_stop, time_cost, departure.arrival_time)
66                          q.put(destination, time_cost)
67      return
```

Czas wykonywania obliczeń - 0.560560941696167s

Kod

Pozostała część kodu dostępna jest w repozytorium:
https://github.com/matto0O/Artificial-Intelligence/tree/master/Lista%201

Problemy i wnioski

Największym problemem okazała się implementacja poprawnie działającej wartości heurystyki, gdyż bez odpowiedniego mnożnika, może dojść do niepotrzebnego przeszukiwania dodatkowych węzłów, zaś przy zbyt dużej wartości, otrzymujemy sytuację, gdy algorytm szuka najkrótszej, a nie najszybszej drogi. Poza tym algorytm A* znacząco przyspieszył działanie programu, bo o około 8%.