Sztuczna inteligencja i inżynieria wiedzy (L)

Adam Pawłowski
266888

Sprawozdanie do listy 1.

Zad 1
Zadanie wykonałem w c++, czego później żałowałem ze względów łatwości obsługi struktury grafu, lecz mówiłem sobie że skoro tyle już zrobiłem to mogę kontynuować. Zmieniłem technologię na pythona w zadaniu 2. z tego właśnie względu. **Nie wziąłem pod uwagę możliwości przesiadki z autobusu np o godzinie 25:00 na autobus o 01:01.** Założyłem też zerowy czas na przesiadkę.

Użyte biblioteki:

```
#include    <iostream>
#include    <vector>
#include    <fstream>
#include    <sstream>
#include    <string>
#include    <map>
#include    <set>
#include    <time.h>
#include    <queue>
```

vector, map, set, oraz queue to STLowe struktury danych, którymi się posługiwałem przy rozwiązaniu. Pozostałe moduły to jedynie pomoce, np time.h umożliwił mi obliczenie czasu trwania poszczególnych algorytmów w celach porównawczych.

Zdefiniowałem strukturę do czytania wpisów z pliku:

```
struct entry{
   string id;
   string company;
   string linia;
   int departure_time; // in minutes passed from 00:00
   int arrival_time; // the same
   string start_stop;
   string end_stop;
   string start_stop_lat;
   string start_stop_lon;
   string end_stop_lat;
   string end_stop_lon;
};
```

Oraz funkcje pomocnicze:

```
int string_time_to_my_time(string s)
```

```
void read_entries()
```

```
void read_stops(const string& filename)
```

Ta ostatnia wyczytuje dane z pliku tekstowego którego użyłem jako input w pierwszym zadaniu. Przykładowa struktura pliku:

KARŁOWICE
KRZYKI
757

Dalej mamy tworzenie grafu. Jest on tworzony w następującej strukturze, z pozoru zagmatwanej, lecz obiecuję wytłumaczyć się z jej wyglądu:

```
vector<vector<pair<pair<int, int>, int> > > G; // ((departure time, arrival time),
where?)
```

Jest to wektor wektorów, który będzie listą sąsiedztwa wierzchołków grafu. Każda krawędź to para<para dwóch intów, inta>, które kolejno odpowiadają za: czas odjazdu daną krawędzią, czas przyjazdu na koniec krawędzi, indeks krawędzi.

Jest to oczywiście znaczne uproszczenie modelu z pliku, dlatego też zdefiniowałem dla wygody kilka struktur pomocniczych. Jeśli chodzi o indeks krawędzi, przypisuję go przy wczytywaniu grafu żeby trzymać w pamięci mniej informacji. Później wykorzystuję indeks do odczytywania innych informacji ze struktur pomocniczych:

```
map<string, int> stops;
map<int, string> spots;
map<int, vector<pair<int,int> > > routes;
map<int, pair<double, double> > coords;
map<pair<int, int>, string> edge_to_line;
```

“stops” to mapa nazwy przystanku na indeks, a “spots” to mapa indeksu na nazwę przystanku. Pozostałe mapy działają w podobny sposób, tylko mamy tam do czynienia z parą indeksów, jeśli chcemy pokazać zależność pomiędzy dwoma przystankami. Później przy tworzeniu grafu odpowiednio wypełniam te struktury:

```
            spots[counter] = e.end_stop;
            coords[counter] = make_pair(stod(e.end_stop_lat), stod(e.end_stop_lon));
            stops[e.end_stop] = counter++;
```

```
edge_to_line[make_pair(start, end)] = e.linia;
```

W funkcji

```
void create_graph()
```

Używam również

```
//sort all edges in the scope of each stop
   for(int i = 0; i < G.size(); i++)
       sort(G[i].begin(), G[i].end());
```

Aby później łatwiej odnajdywać się w czasach krawędzi należących do jednego wierzchołka i wyszukiwaniem binarnym znaleźć czasy które nas interesują.

Dalej, mamy klasycznego dijkstrę na kolejce priorytetowej:

```
priority_queue <pair<int,int>, vector <pair<int, int> >, greater<pair<int, int> > >
PQ; // (time, where?)
vector<bool> vis;
vector<int> arrival_time;
```

Wyżej umieściłem struktury danych potrzebne dijkstrze - tablica odwiedzeń, czasów, oraz samą kolejkę. Kolejność priorytetu jest odwrócona, a samym elementem kolejki to para (czas, dokąd). Czas musi być pierwszy, bo kolejka par będzie sortowana po pierwszym elemencie pary.

```
void dijkstra(string starting_node, int starting_time){
   cleanup();

   arrival_time[stops[starting_node]] = 0;
   PQ.push(make_pair(starting_time, stops[starting_node]));

   while(!PQ.empty()){
       int current_node = PQ.top().second;
       int current_time = PQ.top().first;
       PQ.pop();

       if(current_node == stops[stop_stop])
           break;

       if(!vis[current_node]){
           vis[current_node] = true;

           // consider only connections that are not in the past. thanks to this,
we dont have to check for those later.
           vector<pair<pair<int, int>, int> >::iterator it =
lower_bound(G[current_node].begin(), G[current_node].end(),
make_pair(make_pair(current_time, 0), 0));

           for(; it != G[current_node].end(); ++it){
               pair<pair<int, int>, int> neighbouring_node = *it;   // ((departure
time, arrival time), where?)
               int dep_time = neighbouring_node.first.first;
               int arr_time = neighbouring_node.first.second;
               int neighbouring_stop = neighbouring_node.second;

               if(!vis[neighbouring_stop]){
                   if(arrival_time[neighbouring_stop] > arr_time){
                       arrival_time[neighbouring_stop] = arr_time;
                       PQ.push(make_pair(arr_time, neighbouring_stop));
                       routes[neighbouring_stop].push_back(make_pair(arr_time,
current_node));
                   }
               }
           }
       }
   }
}
```

Astar jest analogiczny, skomplikowała się tylko struktura kolejki, aby dodać f-value po którym sortujemy, potrzebne do algorytmu:

```
priority_queue<pair<int, pair<int, int> >, vector<pair<int, pair<int, int> > >,
greater<pair<int, pair<int, int> > > > PQ_astar; // (f-value, (time, where?))
```

Oraz dodana jest funkcja która determinuje przybliżoną odległość, metryką manhattan albo euklidesową:

```
double heuristic(int current_node, int destination_node, bool manhattan = false)
{
   double dx = coords[current_node].first - coords[destination_node].first;
   double dy = coords[current_node].second - coords[destination_node].second;
   return sqrt(dx * dx + dy * dy);
}
```

W kodzie który pokazałem jest jednak problem, bo na razie liczymy wyniki jedynie pod kryterium czasu. Nie są to piękne drogi, ze względu na ilość przesiadek:

A* time with Manhattan heuristic: 0.011203 s
route KARŁOWICE at 12:37:00 to destination: KRZYKI

KRZYKI at 13:22:00 line 248
Orla at 13:19:00 line 248
Jastrzębia at 13:18:00 line 248
Hallera at 13:17:00 line 257
Sztabowa at 13:15:00 line 257
Rondo at 13:14:00 line 257
Wielka at 13:13:00 line 257
Zaolziańska at 13:12:00 line 23
Arkady (Capitol) at 13:10:00 line 602
Renoma at 13:08:00 line 602
Narodowe Forum Muzyki at 13:05:00 line 602
Rynek at 13:03:00 line 248
Uniwersytet Wrocławski at 13:00:00 line 248
Dubois at 12:58:00 line 250
Paulińska at 12:56:00 line 250
DWORZEC NADODRZE at 12:52:00 line 908
Trzebnicka at 12:50:00 line 16
KARŁOWICE at 12:50:00
Line changes: 8

Ale są szybkie. Zmodyfikujmy jednak strukturę kolejki aby dało się sprawdzić jaką linią dojechaliśmy na przystanek, żeby stwierdzić, czy musimy się przesiadać:

```
priority_queue<pair<pair<int, int>, pair<int, int> >, vector<pair<pair<int, int>,
pair<int, int> > >, greater<pair<pair<int, int>, pair<int, int> > > >
PQ_astar_lines; // ((f-value, last stop), (time, where?))
```

Dodajmy jeszcze wagę 0 jeśli jesteśmy na tej samej linii jako f-value oraz wagę 1 jeśli zmieniamy linię i teraz mamy gotowy algorytm pod kryterium przesiadkowe:

```
// if edge is from the same line, we add 0 to the f-value, otherwise 1

                    if(edge_to_line[make_pair(current_node, neighbouring_stop)]
== edge_to_line[make_pair(previous_node, current_node)])
                        PQ_astar_lines.push(make_pair(make_pair(0,
current_node), make_pair(arr_time, neighbouring_stop)));
                    else
                        PQ_astar_lines.push(make_pair(make_pair(1,
current_node), make_pair(arr_time, neighbouring_stop)));
```

Czasem jednak jest tak, że działa on bardzo na siłę - chce koniecznie nie przesiadać się, nawet jeśli by to było korzystne czasowo. Proponuję więc modyfikację, która bierze pod uwagę nie tylko przesiadki, ale również odległość od celu:

```
// if edge is from the same line, we add 0 to the f-value, otherwise 1
                        double penalty = 1.0 / heuristic(neighbouring_stop,
stops[stop_stop]);



                        if(edge_to_line[make_pair(current_node, neighbouring_stop)]
== edge_to_line[make_pair(previous_node, current_node)])
                            PQ_astar_lines.push(make_pair(make_pair(0 - penalty,
current_node), make_pair(arr_time, neighbouring_stop)));
                        else
                            PQ_astar_lines.push(make_pair(make_pair(0 + penalty,
current_node), make_pair(arr_time, neighbouring_stop)));
```

I mamy znacznie lepsze wyniki. Porównanie dla losowego przypadku dla algorytmów Dijkstry, A* różnymi metrykami, A* pod względem kryterium przesiadek i A* z modyfikacją :

Dijkstra time: 0.011081 s

route KARŁOWICE at 12:37:00 to destination: KRZYKI

KRZYKI at 13:22:00 line 248
KARŁOWICE at 12:50:00
Line changes: 8

A* time: 0.011339 s
route KARŁOWICE at 12:37:00 to destination: KRZYKI

KRZYKI at 13:22:00 line 248
KARŁOWICE at 12:50:00
Line changes: 8

A* time with Manhattan heuristic: 0.011203 s
route KARŁOWICE at 12:37:00 to destination: KRZYKI

KRZYKI at 13:22:00 line 248
KARŁOWICE at 12:50:00
Line changes: 8

A* time with least lines heuristic 0-1: 0.004616 s
route KARŁOWICE at 12:37:00 to destination: KRZYKI

KRZYKI at 13:49:00 line 602
KARŁOWICE at 12:50:00
Line changes: 7

A* time with least lines heuristic: 0.007233 s
route KARŁOWICE at 12:37:00 to destination: KRZYKI

KRZYKI at 13:49:00 line 602
KARŁOWICE at 12:50:00
Line changes: 4

Tutaj wspomnieć muszę o jedynej, moim zdaniem, zalecie pisania tego wszystkiego w c++, czyli wyjątkowo niskich czasach działania algorytmów, które mimo wszystko mają złożoności O(n log n).

Po konsultacji z Panią Prowadzącą, zauważona została dziwna właściwość: linia 253, która jest linią nocną, pojawiała się na trasie o godzinie 9:00. Zlokalizowałem problem w strukturze

```
map<pair<int, int>, string> edge_to_line;
```

Której używam do odczytu drogi później. Oczywiście problemem jest to, że nie ma tutaj wystarczającej informacji w kluczu - pomiędzy dwoma id przystanków będzie wiele linii, które je łączą.

```
// create graph
G.resize(counter + 1);
for (int i = 0; i < entries.size(); i++){
    entry e = entries[i];
    int start = stops[e.start_stop];
    int end = stops[e.end_stop];
    G[start].push_back(
        make_pair(
            make_pair(
                e.departure_time,
                e.arrival_time
            ),
            end
        )
    );
    edge_to_line[make_pair(start, end)] = e.linia;
}
```

Prosta zmiana dodania godziny, choć lepsza, będzie również zbyt słaba bo też możemy mieć dwie linie które w tym samym momencie odjeżdżają z przystanku. Oczywiście nie byłby to problem Dijkstry który patrzy tylko na czasy, ale już zdecydowanie problem kryterium przesiadkowego. Być może jednak jest mało takich przypadków, postanowiłem to sprawdzić, zliczając jak bardzo sytuacja zła była w starym rozwiązaniu i jak zła jest w rozwiązaniu dodającym do klucza czas.

```
int etl_conflict = 0;
int etl2_conflict = 0;

// create graph
G.resize(counter + 1);
for (int i = 0; i < entries.size(); i++){
    entry e = entries[i];
    int start = stops[e.start_stop];
    int end = stops[e.end_stop];
    G[start].push_back(
        make_pair(
            make_pair(
                e.departure_time,
                e.arrival_time
            ),
            end
        )
    );
    if(edge_to_line[make_pair(start, end)] != "")
        etl_conflict++;
    edge_to_line[make_pair(start, end)] = e.linia;

    if(edge_to_line2[make_pair(make_pair(e.departure_time, e.arrival_time), make_pair(start, end))] != "")
        etl2_conflict++;
    edge_to_line2[make_pair(make_pair(e.departure_time, e.arrival_time), make_pair(start, end))] = e.linia;
}
cout << "etl_conflict: " << etl_conflict << endl;
cout << "etl2_conflict: " << etl2_conflict << endl;
exit(0);
```

```
~/Downloads/sztuczna_inteligencja/lista1/zad1 git:(main)±14 (14.024s)
./run
etl_conflict: 994177
etl2_conflict: 582796
```

Jak przewidywałem jest lepiej, ale dalej jest bardzo źle. Z 99% konfliktujących wpisów wartość zmalała jedynie do 60%. Muszę znaleźć inne rozwiązanie.

Rozwiązanie jest proste, ale bardzo mozolne. Potrzebuję jednoznaczną informację jaką linią jadę w danej krawędzi - więc dołączę linię do struktury grafu:

Przed:

```
vector<vector<pair<pair<int, int>, int> > > G; // ((departure time, arrival time), where?)
```

Po zmianach:

```
vector<vector<pair<pair<int, int>, pair<int, string> > > > G; // ((departure time, arrival time), (where?, line))
```

Teraz czekał mnie wielki refactor. Przy refactorze, zobaczyłem że mam krytyczny błąd w strukturze kolejek priorytetoweych algorytmów A* - wrzucałem wartość zmiennoprzecinkową **float** do **inta**, więc wartości były praktycznie zawsze takie same i sortowanie **nie odbywało się** po pierwszej wartości pary par, która prawie zawsze była taka sama, lecz po drugiej, w której był czas. W rezultacie czasowo algorytmy te działały jak Dijkstra (odległość nie miała znaczenia bo była zawsze taka sama, a kryterium czasu miało znaczenie 99% czasu). Jeśli chodzi jednak o ilość przesiadek, wartości te były, niestety, wyssane z palca, przez błąd jaki miałem w mojej strukturze danych “edge_to_line”. Poza poprawką grafu, refactor obejmował też poprawkę struktur kolejek, które teraz miały w sobie odpowiednie typy **double**:

```
priority_queue<pair<double, pair<int, int> >, vector<pair<double, pair<int, int> > >, greater<pair<double, pair<int, int> > > > PQ_astar; // (f-value, (time, where?))
priority_queue<pair<pair<double, string>, pair<int, int> >, vector<pair<pair<double, string>, pair<int, int> > >, greater<pair<pair<double, string>, pair<int, int> > > > PQ_astar_lines; // ((f-value, line-we-arrived-with), (time, where?))
```

Porównanie wyników sprzed i po refactora zamieściłem w tabeli poniżej:

route KARŁOWICE at 12:37:00 to destination: KRZYKI

algorytm	Dijkstra	A*euklides	A*manhattan	A*(0-1)	A*(final)
linie przed	8	8	8	7	4
linie po	4	7	7	6	5
czas przed	32	32	32	59	59
czas po	32	59	59	58	62

Analiza wyników:
linie po: Co zaskakujące, najlepszym algorytmem w kryterium przesiadkowym w tym wypadku wyszedł… algorytm Dijkstry. W algorytmach A* różnych wersji widać jednak tendencję spadkową w ilości linii, więc uznaję to za sukces przynajmniej w tym aspekcie.
czas po: Dijkstra królem, tak jak przewidywano. Pozostałe są bardzo porównywalne, choć różne, co dodaje autentyczności wynikom.

Rząd “linie przed” nie ma sensu być analizowanym, gdyż przez mój poprzedni błąd były tam losowe wartości. Czas przed, w przypadku algorytmów A* przy kryterium czasowym był również naciągany, bo to tak na prawdę był algorytm Dijkstry.

Testując inne przypadki, większość razy Dijkstra nie jest jednak najlepszym algorytmem jeśli chodzi o kryterium przesiadek. Inny przykład, droga Babimojska at 07:00:00 to destination: FAT:

Dijkstra time: 0.004102 s
FAT at 07:13:00 line 124
Babimojska at 07:01:00
Line changes: 4

A* time: 0.000663 s
FAT at 07:15:00 line 607
Babimojska at 07:01:00
Line changes: 3

A* time with Manhattan heuristic: 0.000658 s
FAT at 07:15:00 line 607
Babimojska at 07:01:00
Line changes: 3

A* time with least lines heuristic 0-1: 0.013438 s
FAT at 07:48:00 line 125
Babimojska at 07:01:00
Line changes: 6

A* time with least lines heuristic: 0.000799 s
FAT at 07:15:00 line 607
Babimojska at 07:01:00
Line changes: 3

algorytm	Dijkstra	A*euklides	A*manhattan	A*(0-1)	A*(final)
linie	4	3	3	6	3
czas	13	15	15	48	15

Tutaj widzimy pięknie to, czego się spodziewaliśmy. Dijkstra jest najszybszy, kosztem jednego przystanku, a A*final znalazł najlepszą drogę pod względem przesiadek.

Należy postawić jednak pytanie: **dlaczego tak źle poszło A* (0-1)?**
Jest to najbardziej prymitywny z algorytmów. Jego jedynym kryterium jest to, czy się przesiadamy czy nie. Jako że wagi są dodawane, a przy niezmienianiu linii dodawane jest 0 czyli neutralny element dodawania, gdy np wsiądziemy sobie w pierwszą lepszą linię, to na szczycie kolejki ciągle dodawane będą te krawędzie, które są tej samej linii, niezależnie czy przybliżają nas do celu, czy nie. Tym sposobem “zajmiemy” wiele wierzchołków, tracąc rozwiązania, które mogłyby być lepsze czasowo, oraz ironicznie, przesiadkowo. Opisałem niżej przypadek, w którym będzie on najlepszy, ale nie jest to rezultat który jest powtarzalny.

Jeszcze jeden przypadek zamieszczam poniżej.
Dijkstra time: 0.007438 s
PL. GRUNWALDZKI at 10:16:00 line D
Młodych Techników at 10:02:00
Line changes: 2

A* time: 0.000909 s
PL. GRUNWALDZKI at 10:37:00 line 19
Młodych Techników at 10:02:00
Line changes: 3

A* time with Manhattan heuristic: 0.000884 s
PL. GRUNWALDZKI at 10:37:00 line 19
Młodych Techników at 10:02:00
Line changes: 3

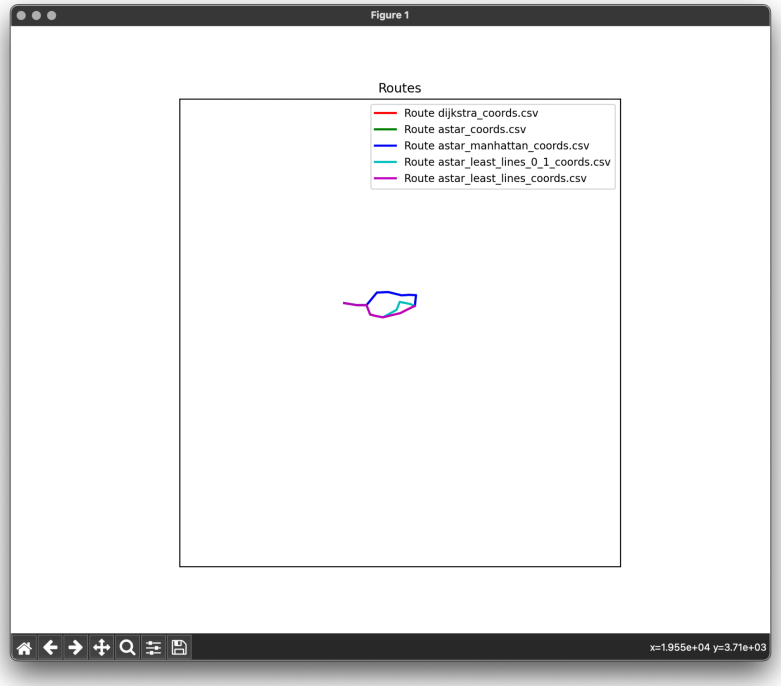
A* time with least lines heuristic 0-1: 0.001058 s
PL. GRUNWALDZKI at 10:19:00 line 10
Młodych Techników at 10:02:00
Line changes: 1

A* time with least lines heuristic: 0.000977 s
PL. GRUNWALDZKI at 10:16:00 line D
Młodych Techników at 10:02:00
Line changes: 3

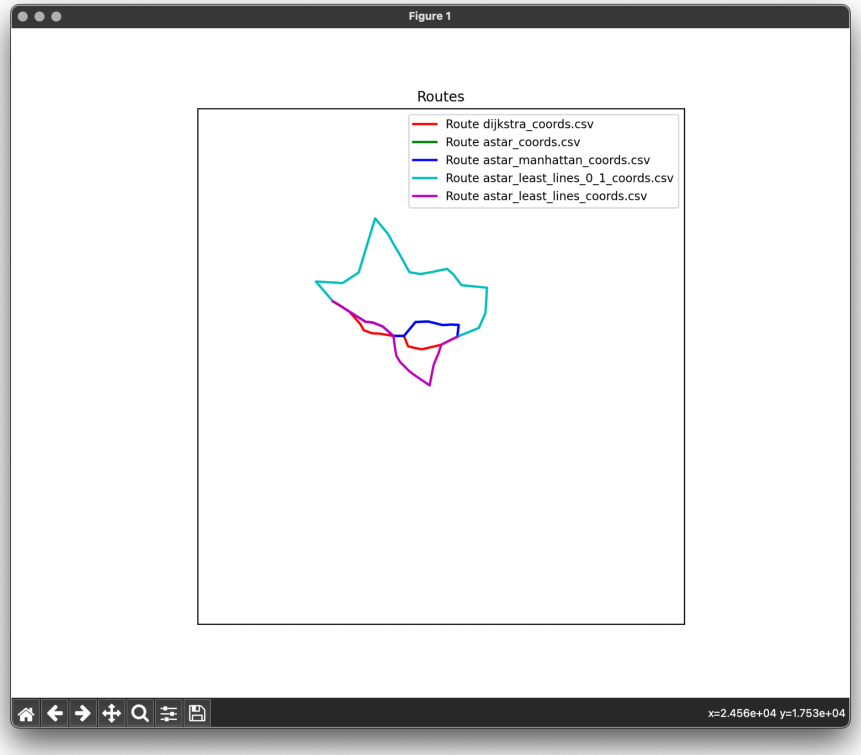
algorytm	Dijkstra	A*euklides	A*manhattan	A*(0-1)	A*(final)
linie	2	3	3	1	3
czas	16	37	37	19	16
czas algorytmu	0.007438	0.000909	0.000884	0.001058	0.000977

W tym przypadku na przykład, pod względem samego kryterium przesiadek, A* (0-1) jest lepszy od jego zmodyfikowanej wersji. Znalazł on bezpośrednią ścieżkę jako jedyny, pomimo że istnieje szybsza ścieżka (znaleziona np przed Dijkstrę).

Czas wykonywania algorytmu jest widocznie lepszy w przypadku wszyskich wersji A*, w porównaniu do Dijkstry, nawet blisko 10 razy.



Stworzyłem również program pokazujący jak przebiegają ścieżki na mapie. Powyżej ścieżka Plac Grunwaldzki - Młodych Techników.
Użyłem do tego `mpl_toolkits basemap, pandas i matplotlib.pyplot.`



Inny przykład, na tasie Kwiska - Plac Grunwaldzki.

Zad 2.
Patrząc na rozwiązania kolegów, którzy użyli jupytera do wczytania danych aby móc raz je wczytać i sobie później na nich operować, postanowiłem ukraść im pomysł i zrobić je również w jupyterze. Założenia pozostają te same, tzn: brak przesiadek z 25:00 na 01:01 oraz zerowy czas na przesiadkę.

Użyte biblioteki:
```
import pandas as pd
import datetime
import networkx as nx
import sys
import time as tm
import numpy as np
import random
import sys
```

Importowane biblioteki w skrypcie wykonują następujące funkcje:
1. `pandas as pd` - Obsługuje manipulację danymi w postaci ramki danych.
2. `datetime` - Zapewnia narzędzia do pracy z datami i czasem.
3. `networkx as nx` - Umożliwia analizę i manipulację grafami i sieciami.
4. `sys` - Udostępnia funkcje i narzędzia związane z systemem operacyjnym.
5. `time as tm` - Dostarcza funkcje związane z pomiarowaniem czasu.
6. `numpy as np` - Zapewnia obsługę wielowymiarowych tablic oraz funkcji matematycznych.
7. `random` - Umożliwia generowanie liczb losowych.

Plik wczytuję pandas-em:
```
df = pd.read_csv("connection_graph.csv", low_memory=False)
```

Ucinam moim zdaniem niepotrzebne dane, skoro już używam pandas i jest to bardzo wygodne. W c++ ta sama robota była znacznie bardziej mozolna. Wczytane dane po zmianach:

line	departure_time	arrival_time	start_stop	end_stop	departure_time_seconds	arrival_time_seconds	
id							
0	A	20:52:00	20:53:00	Zajezdnia Obornicka	Paprotna	75120	75180
1	A	20:53:00	20:54:00	Paprotna	Obornicka (Wołowska)	75180	75240
2	A	20:54:00	20:55:00	Obornicka	Bezpieczna	75240	75300

				(Wołowska)			
3	A	20:55:00	20:57:00	Bezpieczna	Bałtycka	75300	75420
4	A	20:57:00	20:59:00	Bałtycka	Broniewskiego	75420	75540

Poźniej tworzę graf używając networkx:

```
# Graph creation
stop_graph = nx.DiGraph()

def create_stop_nodes(conn):
   stop_graph.add_node(conn['start_stop'], lat=conn['start_stop_lat'],
lon=conn['start_stop_lon'])

stops.apply(create_stop_nodes, axis=1)
def create_edges(conn):
   if not stop_graph.has_edge(conn['start_stop'], conn['end_stop']):
       stop_graph.add_edge(conn['start_stop'], conn['end_stop'], schedule=[])

   # Append the new connection data to the schedule list

stop_graph[conn['start_stop']][conn['end_stop']]['schedule'].append(conn.to_dict())

df.apply(create_edges, axis=1)
```

Dalej, kluczowymi funkcjami dla tabu search względem dwóch kryteriów będą funkcje estymujące koszt względem przesiadek i czasu drogi:

```
average_speed_mpk_km_s = 17 / 3600  # average velocity in Wrocław [km/s]
degrees_to_km = 111

def best_connection_min_time(schedule, current_time_seconds, current_cost,
previous_connection=None):
   best_connection = None
   best_cost = float('inf')

   for connection in schedule:
       waiting_time = (connection['departure_time_seconds'] - current_time_seconds
+ DAY_SECONDS) % DAY_SECONDS
       travel_time = (connection['arrival_time_seconds'] -
connection['departure_time_seconds'] + DAY_SECONDS) % DAY_SECONDS
       total_cost = current_cost + waiting_time + travel_time

       if total_cost < best_cost:
           best_cost = total_cost
```

```
            best_connection = connection

    return (best_connection, best_cost)

def best_connection_min_transfers(schedule, current_time_seconds, current_cost,
previous_connection):
    previous_line = ""
    if previous_connection:
        previous_line = previous_connection['line']
    best_connection = None
    best_cost = float('inf')

    for connection in schedule:
        waiting_time = (connection['departure_time_seconds'] - current_time_seconds
+ 86400) % 86400
        travel_time = ((connection['arrival_time_seconds'] -
connection['departure_time_seconds'] + 86400) % 86400)
        total_cost = current_cost + waiting_time + travel_time

        if previous_line and connection['line'] != previous_line:
            total_cost += 1000  # Penalizing for a line change

        if best_cost > total_cost:
            best_cost = total_cost
            best_connection = connection

    return (best_connection, best_cost)
```

Dalej definiuję funkcje samplujące sąsiadów:

```
# Sampling a subset of neighbors for efficiency
def sample_neighbors(neighbors, sample_size=7):
    return random.sample(neighbors, sample_size)

# Generating neighbor solutions by swapping stops
def generate_neighbors(route):
    neighbor_solutions = []
    for i in range(len(route)):
        for j in range(i + 1, len(route)):
            new_route = route[:]
            new_route[i], new_route[j] = new_route[j], new_route[i]
            neighbor_solutions.append(new_route)
    return sample_neighbors(neighbor_solutions)
```

Oraz sam tabu search:

```
def tabu_search_strategy(initial_route, iterations_limit, tabu_size, initial_stop,
departure_time, transport_network,
connection_strategy=best_connection_min_transfers):
   search_start_time = tm.time()

   optimal_route = initial_route
   current_route = initial_route
   tabu_registry = []

   for _ in range(iterations_limit):
       neighbor_routes = generate_neighbors(current_route)
       optimal_neighbor = None
       optimal_neighbor_cost = float('inf')

       # Evaluate only non-tabu neighbors
       for neighbor in neighbor_routes:
           if neighbor not in tabu_registry:
               cost = evaluate_solution_cost(neighbor, initial_stop,
departure_time, transport_network, connection_strategy)
               if cost < optimal_neighbor_cost:
                   optimal_neighbor = neighbor
                   optimal_neighbor_cost = cost

       # Break if no improvement is found
       if optimal_neighbor is None:
           break

       current_route = optimal_neighbor
       tabu_registry.append(optimal_neighbor)
       if len(tabu_registry) > tabu_size:
           tabu_registry.pop(0)

       # Update optimal route if a better solution is found
       if evaluate_solution_cost(optimal_neighbor, initial_stop, departure_time,
transport_network, connection_strategy) < evaluate_solution_cost(optimal_route,
initial_stop, departure_time, transport_network, connection_strategy):
           optimal_route = optimal_neighbor

   search_duration = tm.time() - search_start_time

   return (optimal_route, solution_costs, search_duration)
```

Wywołanie go wygląda na przykład tak:

```
initial_stop = "PL. GRUNWALDZKI"
departure_hour = datetime.time(7, 0, 0)
route_plan = ["Babimojska", "FAT", "Swojczyce", "Biegasa", "Młodych Techników"]
iterations_limit = 6
tabu_size = 8

# Performing tabu search to find the best route solution
(optimal_route, evaluated_solutions, search_time) =
tabu_search_strategy(route_plan, iterations_limit, tabu_size, initial_stop,
departure_hour, stop_graph, connection_strategy=best_connection_min_transfers)
```

Wyniki takiego wywołania dla powyższej ścieżki, dla kryterium przesiadkowego, dla tablicy tabu która nie ma limitu:

```
Best route: ['FAT', 'Babimojska', 'Młodych Techników', 'Biegasa', 'Swojczyce']
Departure:  PL. GRUNWALDZKI Arrival:  FAT
('enter line: D', datetime.time(7, 1), 'at: PL. GRUNWALDZKI')
('leave line: D', datetime.time(7, 11), 'at: Arkady (Capitol)')
('enter line: A', datetime.time(7, 11), 'at: Arkady (Capitol)')
('leave line: A', datetime.time(7, 23), 'at: FAT')
Departure:  FAT Arrival:  Babimojska
('enter line: 126', datetime.time(7, 23), 'at: FAT')
('leave line: 126', datetime.time(7, 31), 'at: Nowodworska')
('enter line: 142', datetime.time(7, 31), 'at: Nowodworska')
('leave line: 13', datetime.time(7, 36), 'at: Babimojska')
Departure:  Babimojska Arrival:  Młodych Techników
('enter line: 13', datetime.time(7, 36), 'at: Babimojska')
('leave line: 13', datetime.time(7, 45), 'at: Młodych Techników')
Departure:  Młodych Techników Arrival:  Biegasa
('enter line: 13', datetime.time(7, 45), 'at: Młodych Techników')
('leave line: 13', datetime.time(7, 55), 'at: GALERIA DOMINIKAŃSKA')
('enter line: 3', datetime.time(7, 56), 'at: GALERIA DOMINIKAŃSKA')
('leave line: 3', datetime.time(8, 6), 'at: Armii Krajowej')
('enter line: 143', datetime.time(8, 6), 'at: Armii Krajowej')
('leave line: 143', datetime.time(8, 10), 'at: Biegasa')
Departure:  Biegasa Arrival:  Swojczyce
('enter line: 143', datetime.time(8, 10), 'at: Biegasa')
('leave line: 143', datetime.time(8, 17), 'at: SĘPOLNO')
('enter line: 115', datetime.time(8, 18), 'at: SĘPOLNO')
('leave line: 115', datetime.time(8, 23), 'at: Swojczyce')
Departure:  Swojczyce Arrival:  PL. GRUNWALDZKI
('enter line: 115', datetime.time(8, 24), 'at: Swojczyce')
('leave line: 115', datetime.time(8, 38), 'at: PL. GRUNWALDZKI')

Total cost:  ('hours: ', 1, 'minutes: ', 38)
Execution time:  434.7344219684601
```

Wyniki dla tablicy tabu o rozmiarze 8:

```
Best route: ['FAT', 'Babimojska', 'Młodych Techników', 'Biegasa', 'Swojczyce']
Departure: PL. GRUNWALDZKI, Arrival: FAT
('enter line: D', datetime.time(7, 1), 'at: PL. GRUNWALDZKI')
('leave line: D', datetime.time(7, 11), 'at: Arkady (Capitol)')
('enter line: A', datetime.time(7, 11), 'at: Arkady (Capitol)')
('leave line: A', datetime.time(7, 23), 'at: FAT')
Departure: FAT, Arrival: Babimojska
('enter line: 126', datetime.time(7, 23), 'at: FAT')
('leave line: 126', datetime.time(7, 31), 'at: Nowodworska')
('enter line: 142', datetime.time(7, 31), 'at: Nowodworska')
('leave line: 13', datetime.time(7, 36), 'at: Babimojska')
Departure: Babimojska, Arrival: Młodych Techników
('enter line: 13', datetime.time(7, 36), 'at: Babimojska')
('leave line: 13', datetime.time(7, 45), 'at: Młodych Techników')
Departure: Młodych Techników, Arrival: Biegasa
('enter line: 13', datetime.time(7, 45), 'at: Młodych Techników')
('leave line: 13', datetime.time(7, 55), 'at: GALERIA DOMINIKAŃSKA')
('enter line: 3', datetime.time(7, 56), 'at: GALERIA DOMINIKAŃSKA')
('leave line: 3', datetime.time(8, 6), 'at: Armii Krajowej')
('enter line: 143', datetime.time(8, 6), 'at: Armii Krajowej')
('leave line: 143', datetime.time(8, 10), 'at: Biegasa')
Departure: Biegasa, Arrival: Swojczyce
('enter line: 143', datetime.time(8, 10), 'at: Biegasa')
('leave line: 143', datetime.time(8, 17), 'at: SĘPOLNO')
('enter line: 115', datetime.time(8, 18), 'at: SĘPOLNO')
('leave line: 115', datetime.time(8, 23), 'at: Swojczyce')
Departure: Swojczyce, Arrival: PL. GRUNWALDZKI
('enter line: 115', datetime.time(8, 24), 'at: Swojczyce')
('leave line: 115', datetime.time(8, 38), 'at: PL. GRUNWALDZKI')

Total cost: ('hours: ', 1, 'minutes: ', 38)
Execution time: 189.98430585861206
```

Widzimy więc, że wyniki są identyczne, poza czasem jaki zajął naszym algorytmom na ich otrzymanie.

Rozmiar 4:

```
Departure: PL. GRUNWALDZKI, Arrival: FAT
('enter line: D', datetime.time(7, 1), 'at: PL. GRUNWALDZKI')
('leave line: D', datetime.time(7, 11), 'at: Arkady (Capitol)')
('enter line: A', datetime.time(7, 11), 'at: Arkady (Capitol)')
('leave line: A', datetime.time(7, 23), 'at: FAT')
Departure: FAT, Arrival: Babimojska
('enter line: 126', datetime.time(7, 23), 'at: FAT')
('leave line: 126', datetime.time(7, 31), 'at: Nowodworska')
('enter line: 142', datetime.time(7, 31), 'at: Nowodworska')
('leave line: 13', datetime.time(7, 36), 'at: Babimojska')
Departure: Babimojska, Arrival: Młodych Techników
('enter line: 13', datetime.time(7, 36), 'at: Babimojska')
('leave line: 13', datetime.time(7, 45), 'at: Młodych Techników')
Departure: Młodych Techników, Arrival: Biegasa
('enter line: 13', datetime.time(7, 45), 'at: Młodych Techników')
('leave line: 13', datetime.time(7, 55), 'at: GALERIA DOMINIKAŃSKA')
('enter line: 3', datetime.time(7, 56), 'at: GALERIA DOMINIKAŃSKA')
('leave line: 3', datetime.time(8, 6), 'at: Armii Krajowej')
('enter line: 143', datetime.time(8, 6), 'at: Armii Krajowej')
```

```
('leave line: 143', datetime.time(8, 10), 'at: Biegasa')
Departure: Biegasa, Arrival: Swojczyce
('enter line: 143', datetime.time(8, 10), 'at: Biegasa')
('leave line: 143', datetime.time(8, 17), 'at: SĘPOLNO')
('enter line: 115', datetime.time(8, 18), 'at: SĘPOLNO')
('leave line: 115', datetime.time(8, 23), 'at: Swojczyce')
Departure: Swojczyce, Arrival: PL. GRUNWALDZKI
('enter line: 115', datetime.time(8, 24), 'at: Swojczyce')
('leave line: 115', datetime.time(8, 38), 'at: PL. GRUNWALDZKI')

Total cost: ('hours: ', 1, 'minutes: ', 38)
Execution time: 10.629596948623657
```

rozmiar tablicy tabu	10000	8	4
czas	434.7344219684601	189.98430585861206	10.629596948623657

Niestety otrzymałem te same wyniki kosztu czasu dla wszystkch rozmiarów tablic. Prawdopodobnie wynika to z faktu, że w rozwiązaniu tabu_searcha używam algorytmu A* do estymacji kosztu ścieżek, i estymacja ta jest bardzo dobra od razu, po kilku pierwszych iteracjach.

Niemniej jednak tabu search został zaimplementowany zgodnie z wytycznymi, i widać po wynikach że ogromne znaczenie czasowe ma dobór rozmiaru tablicy.

Przybliżona trasa: