Sztuczna inteligencja i inżynieria wiedzy

Lista 1

Konrad Kiełczyński 260409

1.Preprocesing danych

Dane z pliku connection_graph.csv najpierw wczytałem za pomocą biblioteki **pandas** następnie usunąłem jeden indeks oraz nazwę firmy których nie używałem. Dane następnie zostały wczytanie do grafu będącego słownikiem o następującej strukturze.

```
graph[start] = {
    "start_pos": set(),
    "next_stop": []
}
```

W kluczu z nazwą **start_pos** był trzymany Set ze wszystkimi Koordynatami powiązanymi z daną nazwą przystanku.

```
namedtuple("Cords", ["X", "Y"])
```

W klucz z nazwa **next_stop** były trzymane wszystkie połączenia wychodzące z danego przystanku z następującą strukturą

```
namedtuple("NextStop", ["name", "cords", "line", "departure", "arrival"])
```

Takie dane zostały skojarzone z konkretnymi polami w tuplach.

```
start_pos = Cords(row["start_stop_lat"], row["start_stop_lon"])
end_pos = Cords(row["end_stop_lat"], row["end_stop_lon"])
next_stop = NextStop(row["end_stop"], end_pos, row["line"],
                     util.convert_to_seconds(row["departure_time"]),
                     util.convert_to_seconds(row["arrival_time"]))
```

Dane koordynatów konkretnego przystanku zostały uśrednione oraz przypisane jak jedne koordynaty

W algorytmach przyjmuję że nie potrzebny jest czas na zmianę połączenia.

2. Dijkstra algorithm

Algorytm Dijkstry to popularny algorytm do znajdowania najkrótszej ścieżki w grafie ważonym, w naszym grafie wagę przyjmuje jako **neighbor.arrival – time** czyli czas dostania się do przystanku o nazwie **neighbor.name**.

Algorytm polega na stopniowym rozszerzaniu najkrótszych ścieżek z wierzchołka startowego do innych wierzchołków grafu, aż do osiągnięcia wierzchołka docelowego. Te wierzchołki filtrujemy by znaleźć te na które zdążymy czyli **neighbor.departure >= curr_dist + time** są one zapisywane do generatora przez list comprehension a nie do sticte listy by przyspieszyć troszkę algorytm.

Premiuję poruszanie się tą samą linią czyli jeżeli uzyskamy taką samą wagę poruszając się tą samą linią wybieramy ją.

3. A* (star) with time optimalization

Algorytm A* to popularny algorytm do wyszukiwania najkrótszej ścieżki między dwoma punktami na grafie. Algorytm ten wykorzystuje funkcję heurystyczną, która estymuje koszt przejścia z

aktualnego węzła do celu co pozwalana na znalezienie oraz wybór najlepiej prosperującej ścieżki. Dzięki temu możemy znaleźć optymalne (często nie najlepsze) rozwiązanie bez konieczności przeszukania całego grafu tak jak w algorytmie Dijkstry.

W algorytmie stosuję **listę otwartych oraz zamkniętych** przystanków/ wierzchołków. Lista otwartych jest to kolejka priorytetowa oparta na kopcu binarnym z wartością funkcji kosztu oraz heurystyki. Została ona użyta by nie sortować za każdym razem listy otwartych, ma to większą złożoność niż odbudowa kopca. Przy potrzebje dostania się za każdym razem do node z najmniejszą wartością dostajemy go szybciej. Wykorzystuję kopiec z **biblioteki heapq**. Stosuję także listę zamkniętych by nie wchodzić i rozwijać tych samych węzłów kilku krotnie.

W algorytmie stosuję też filtrowanie listy rozwijanej o połączenia na które jesteśmy wstanie zdążyć oraz te które są już extended(zamknięte).

W przypadku mojego rozwiązanie, stosuję heurystykę odległości **Haversine** między danym węzłem a węzłem końca oraz koszt **neighbor.arrival – time** . Ta heurystyka oblicza odległość między dwoma punktami na sferze ziemskiej jako iloczyn odległości kątowej i promienia Ziemi w kilometrach. Następnie mnożę tą wartość przez 250, aby dostosować ją do zakresu wartości używanych w algorytmie. Użyłem do liczenia tej odległości **biblioteki havestine** oraz funkcji z niej o takiej samej nazwie.

4. A* (star) with line change optimalization

W tym algorytmie starałem się zoptymalizować liczbę przesiadek do tego wykorzystałem filtrowanie sąsiadów oraz heurystykę premiującą linie które są wspólne z liniami docierającymi do naszego końca. Informacje o **liniach wspólnych z celem** trzymam w słowniku by mieć do tej informacji dostęp liniowy.

```
goal_lines_dict = {x.line: True for x in graph[goal]["next_stop"]}
```

W algorytmie stosuję też filtrowanie listy sąsiadów czyli połączenia na które jesteśmy wstanie zdążyć oraz te które są już extended(zamknięte) tak jak w poprzednim przykładzie.

Jako koszt rozwiązania uznałem sumowany **koszt przesiadek** gdzie jedna przesiadka równa się 250, oraz **koszt czasu** jak w poprzednim rozwiązaniu by nie było sytuacji że czekamy na linię która jest przystankiem końcowym ale jest za 5h(nocne, przy testach tak się zdarzyło :)). Te 2 wartości tworzyły koszt który trzymałem w tuple.

W heurystyce **linie końcowe** premiowałem priorytetem równym 0 a odwrotnie 900. Podobnie premiowałem przesiadanie w **centrach przesiadkowych** zrobiłem to w taki sposób że liczyłem różnicę od największego centrum z 2806 połączeniami odejmowałem połączenia wychodzące z sąsiada i to dzieliłem przez 10. Te wartości zsumowane tworzyły heurystykę.

5. Modyfikacja algorytmu A *

Do modyfikacji algorytmu wybrałem algorytm minimalizujący czas przejazdu, do tego spróbowałem wybrać heurystykę która wykonuje się jak najszybciej dzięki temu algorytm też będzie szybszy.

Pod uwagę wziąłem 6 heurystyk włącznie z tą którą wybrałem na początku. Heurystyki które

badałem. Te heurystyki mogły zostać użyte ze względu na to że nie bierzemy zakrzywienia ziemi na tak małym wycinku rzeczywistości

- Haversine distance: oblicza odległość między dwoma punktami na sferze ziemskiej jako iloczyn odległości kątowej i promienia Ziemi.
- Manhattan distance: oblicza sumę bezwzględnych różnic między odpowiadającymi sobie współrzędnymi punktów.
- Euclidean distance: oblicza odległość między dwoma punktami w przestrzeni, wykorzystując twierdzenie Pitagorasa.
- Unidimensional distance: oblicza różnicę między dwoma punktami w jednym wymiarze, wykorzystując wartość maksymalną modułu różnic pomiędzy odpowiadającymi sobie współrzędnymi.
- Cosine distance: oblicza podobieństwo kosinusowe między dwoma wektorami.
- Chebyshev distance: oblicza maksymalną różnicę między odpowiadającymi sobie współrzędnymi punktów.

Dane heurystyki pomnożyłem przez obliczone wcześniej wartości by dawały miary podobne do tych co Haversine. Następnie puściłem 10 000 losowych połączeń tych samych dla każdej z heurystyk czyli łącznie 60 000 co trwało około 30 minut.
Wyniki przedstawiają się następująco :

Nazwa heurystyki avg czas avg cost w sekundach

```
cosine_distance          : 0.01451330359 : 8738.413017276014
chebyshev_distance       : 0.02446869937 : 6359.993996998499
unidimensional_distance  : 0.02451174706 : 6359.993996998499
euclidean_distance       : 0.02986139946 : 6262.4032016008005
manhattan_distance       : 0.03255361437 : 6190.63731865933
haversine_distance       : 0.03408822835 : 6188.086043021511
```

Po uzyskaniu wyników więc wybieram **cosine_distance** czyli podobieństwo kosinusowe między dwoma wektorami. Okazało się że ta heurystyka najbardziej optymalizuje czas wykonywania całego algorytmu oraz w porównaniu z innymi bije je na głowe. Jednak niezbyt radzi sobie z optymalizacją kosztu gdy bierzemy to też pod uwagę to najlepszy jest **chebyshev distance** który średnio znajduje rozwiązanie o 2 minuty większe niż najlepsze haversine.

6. Tabu search

Tabu serach to algorytm służący do rozwiazywania algorytmów optymalizacyjnych, celem jego jest znalezienie rozwiązania w oparciu o iteracyjne przeszukiwanie. W kolejnych iteracjach algorytm przeszukuje sąsiednie rozwiązania i wybiera najlepsze z nich jako najlepsze rozwiązanie. Zmianę doprowadzającą do tej zmiany zapisuje do listy tabu. Na tej liście, znajdują się ruchy, które są zakazane w najbliższych iteracjach. Dzięki temu algorytm tabu search ma większe szanse na znalezienie globalnego minimum lub maksimum, ponieważ unika wpadnięcia w minimum lokalne co jest częstym problemem.

W tym zadaniu tak naprawdę rozwiązujemy problem Komiwojażera w którym musimy odwiedzić wszystkie przystanki zaczynając od przystanku początkowego i kończąc także na nim. Do algorytmu

dodajemy maksymalną liczbę iteracji oraz ile maksymalnie iteracji może być wykonanych bez poprawy rozwiązania.

```
max_iterations = math.ceil(1.1 * (n_stops ** 2))
turns_improved = 0
improve_thresh = 2 * math.floor(math.sqrt(max_iterations))
```

Na początku najlepsze rozwiązanie jest losowane ponieważ od czegoś musimy zacząć. Następnie zaczynamy iteracje i przeszukujemy wszystkich sąsiadów, po tym mamy aktualne i najlepsze rundy. Następnie dodajemy ten ruch na listę tabu. Jeżeli ta wartość jest lepsza od najlepszego rozwiązania to podmieniamy ją gdy jednak nie jest inkrementujemy iterację bez poprawy.

- Modyfikacja z ograniczeniem listy tabu

Długość tablicy tabu ma wpływ na to, jak wiele ruchów jest blokowanych w kolejnych iteracjach algorytmu. Im dłuższa lista tabu, tym więcej ruchów jest blokowanych, co może pomóc w uniknięciu minimum lokalnego, ale może też spowodować zbyt szybkie zablokowanie ruchów, które są potrzebne do znalezienia najlepszego rozwiązania.
Po kilku próbach ustawiłem listę tabu na długość wszystkich stopów na drodze

```
tabu_tenure =  n_stops
```

- Modyfikacja o aspiracje w celu minimalizacji funkcji kosztu

Aspiracja jest mechanizmem w algorytmie Tabu Search, który pozwala na zignorowanie zakazu ruchu, który normalnie znajduje się na liście tabu. W przypadku, gdy dany ruch prowadzi do znalezienia lepszego rozwiązania niż aktualnie najlepsze, aspiracja umożliwia wykonanie tego ruchu pomimo jego obecności na liście tabu. W praktyce, mechanizm aspiracji w Tabu Search działa na zasadzie warunku akceptacji, który sprawdza, czy dane rozwiązanie jest wystarczająco dobre, aby zignorować jego obecność na liście tabu.

Aspiracje ustawiam jako wartość połowy aktualnie najlepszego rozwiązania co ustawia tą wartość na dynamiczną podczas działania programu

```
aspiration_criteria = best_solution_cost * 0.5
```

- Rozszerzenie poprzez próbkownaie sąsiedztwa

Dobór strategii próbkowania sąsiedztwa pozwala na skrócenie czasu potrzebnego do znalezienia optymalnego rozwiązania oraz wykonania większej ilości iteracji w tym samym czasie. Jak i wybór odpowiedniej strategii może również pomóc w uniknięciu wpadania w lokalne minima i zwiększyć szansę na znalezienie lepszego rozwiązania.

Można wybrać np. zmianę 2 losowych przystanków na trasie 2-opt i modyfikacje n-opt jak i łączenie ich
https://en.wikipedia.org/wiki/2-opt

Kolejne rozwiązanie polegające na odwróceniu losowego podzbioru jednak jest one bardziej złożone.

Wybieram rozwiązanie 2-opt ponieważ jest podobne do już zaimplementowanego. Oraz aby skrócić czas zminiejszę liczbę takich zmian do n_stops a nie przeszukiwania wszystkich możliwości

Czas po tej modyfikacji znacząco się zmienił z 0.0268s na 0.0059s podczas gdy rozwiązanie uległo nawet małemu polepszeniu. Testowane było na jednym randomowym seedzie

Nowe rozwiązanie

```
for i in range(math.floor(0.8 * n_stops)):
    neighbor, move = two_opt(current_solution[:])
```

```
def two_opt(current):
    prev = random.randint(0, len(current) - 1 )
    curr = random.randint(0, len(current) - 1)
    current[curr], current[prev] = current[prev], current[curr]
    return current, (prev, current)
```

Stare rozwiązanie

```
    for i in range(n_stops):
        for j in range(i + 1, n_stops):
            neighbor = current_solution[:]
            neighbor[i], neighbor[j] = neighbor[j], neighbor[i]
```

Podsumowanie

Głównym problem tego zadania były dane które często dawały dziwne nieoczekiwane rezultaty np. był jeden przystanek o nazwie Żórawina - Niepodległości (Mostek) oraz kilka przed nim z którego nie dało się wyjechać. Także ciężko było osiągnąć na samym A* optymalny wynik z liniami bez pomieszania z innymi algorytmami przeszukiwania oraz jakąś większą wiedzą dotyczącą linii.

Solution:
https://github.com/Golden3x11/AI/tree/main/searching_a_star_djikstra