Sztuczna inteligencja i inżynieria wiedzy

laboratorium - Lista 1

Problemy optymalizacyjne

Vera Yemialyanava, 269565

Wrocław, 22.03.2024

Contents

1 Zadanie 1

1.1 Przygotowanie danych

Do wczytania danych z pliku connection_graph.csv została utworzona funkcja load_data() w modułu *load_data*. Wczytanie pliku .csv zrobiono za pomocą funkcji z biblioteki pandas. W tej funkcji iterujemy przez każdy wiersz wczytanych danych i przechowujemy dane w docelowych zmiennych, a mianowicie: **start** i **end** to przystanki początkowe i końcowe połączenia, **start_pos** i **start_pos** reprezentują współrzędne geograficzne tych przystanków w postaci klasy Cords, **next_stop** reprezentuje następny przystanek w postaci klasy NextStop.

```
Cords = namedtuple("Cords", ["X", "Y"])
NextStop = namedtuple("NextStop", ["name", "cords", "line", "departure", "arrival"])
```

Następnie został utworzony graf w postaci słownika, gdzie kluczem jest nazwa przystanku.

```
"start_pos": set(),
"next_stop": set()
```

Dalej do każdego przystanku tworzono słownik, zawierający dwa klucze: start_pos ze wszystkimi współrzędnymi powiązanymi z daną nazwą przystanku oraz next_stop, gdzie trzymamy wszystkie połączenia wychodzące z danego przystanku. Dla każdego przystanku początkowego i końcowego tworzony jest wpis w grafie (jeśli jeszcze nie istnieje). W obrębie tej implementacji przyjęłam, że czas potrzebny na zmianę połączenia nie jest uwzględniany. Dane współrzędnych geograficznych zostały uśrednione.

1.2 Dijkstra: kryterium czasu

Algorytm Dijkstry jest algorytmem do znalezienia najkrótszej ścieżki od pewnego wierzchołka do pozostałych w grafie ważonym. W naszym przypadku wagą krawędzi jest czas dojazdu od przystanku do jego siąsiadu. Do implementacji algorytmu Dijkstra w modulu *dijkstra* stworzono funkcje do znalezienia najszybszego połączenia pomiędzy przystankiem startowym i pozostałymi przystankami:

```
def dijkstra(graph_dict, start, time):
```

Bierzemy wierzchołek curr_node o najmnijszym czasie dojazdu curr_dist z kolejki priorytetowej, tym samym rozpatrywany wierzchołek o najmniejszym czasie. Wykorzystano modul heapq do implementacji działania kolejki priorytetowej. Dla danego wierzchołka ustalamy, z której linii pochodzi. Dalej tworzymy generator dla wszystkich potencjalnych połączeń (next_stop) z bieżącego wierzchołka i uwzględniamy tylko przystanki na które zdążymy:

```
filtered_stops = (n for n in graph_dict[curr_node]["next_stop"] if n.departure >= curr_dist + time)
```

Zapewnia to, że rozważane są tylko realistyczne połączenia, dostępne po dotarciu do przystanku. Dalej dla każdego połączenia aktualizujemy czas dojazdu. Czas dojazdu obliczamy jako różnicę pomiędzy czasem przyjazdu a naszym czasem początkowym:

```
new_dist = neighbor.arrival - time
```

Poruszanie się tę samą linią ma wyższy priorytet w przypadku tych samych wartości wag. Przykładowe wyniki:

1.3 A*: kryterium czasu

Algorytm A* jest algorytmem do znalezienia najkrótszej ścieżki między dwoma wierzchołkami. Znalezienie rozwiązania nie wymaga przeszukania całego grafu tak jak w algorytmie Dijkstry. Natomiast otrzymane wyniki nie zawsze są najbardziej optymalnymi, ale zazwyczaj są blisko docelowych wartości. Jest to możliwe dzięki heurystykom - metodom poszukiwania dobrych rozwiązań przybliżonych. Heurystyki są funkcjami, które estymują koszt przejścia z aktualnego węzła do celu.

```
priority_queue = [(0, start)]  # lista otwartych węzłów
visited = set()  # zbiór zamkniętych węzłów
```

Wykorzystano kolejkę priorytetową, zawierającą węzły do rozważenia, która umożliwia nam szybsze uzyskanie węzła z najmniejszym kosztem oraz zbiór węzłów już przetworzonych, by nie analizować ich wielokrotnie. Zmienne came_from i cost_so_far służą do przechowywania danych o najlepszych poprzednikach każdego węzła na ścieżce oraz koszt dotarcia do każdego węzła. Kosztem w tym przypadku jest czas. Tak samo jak i w poprzedniej implementacji uwzględniamy tylko te przystanki, na które zdążymy:

```
filtered_stops = (n for n in graph_dict[curr_node]["next_stop"] if n.departure >= curr_dist + time)
```

Jako funkcję heurystyki została użyta funkcja distance z geopy.distance. Oblicza ona odległości pomiędzy punktami na podstawie formuły Vincenty. Ponieważ dane używane do określenia priority w algorytmem są w postaci sekund i wynoszą wartości rzędu tysięcy zdecydowano pomnożyć odległość otrzymaną w kilometrach z funkcji heurystyki przez 1000. Ma to na celu dostosować wartości do siebie. Jak i w poprzedniej implementacji czas dojazdu jest obliczany jako różnica pomiędzy czasem przyjazdu a naszym czasem początkowym.

```
new_dist = neighbor.arrival - time
```

1.4 A*: kryterium przesiadek

W tym algorytmie jak i w poprzedniej implementacji wykluczamy połączenia, na które nie zdążymy z obecnej lokalizacji biorąc pod uwagę bieżący czas oraz połączenia już rozpatrzone, aby uniknąć ponownego analizowania tych samych tras. Jak i w poprzedniej implementacji przechowujemy koszt naszego rozwiązania. Jednak w tym przypadku kosztem naszego rozwiązania jest sumaryczna liczba przesiadek. Do kosztu zdecydowano dodać czas aby pozbawić się dużych czasów oczekiwania na linię.

```
priority = new_cost + new_time / 3 + heuristic_fn
```

W celu szybkiego sprawdzenia, czy dany przystanek należy do linii prowadzących bezpośrednio do celu wprowadzono słownik goal_line_dict. W tym słowniku ustawiamy flagę true dla linii, które sąsiadują z przystankiem docelowym. Każda przesiadka wynosi koszt równy 1000.

```
new_cost = (0 if line == neighbor.line else 1000) + current_cost
```

Funkcja heurystyki sprawdza, czy linia aktualnie rozpatrywanego sąsiada prowadzi do celu i dodaje "karne" 1000 punktów w przypadku braku spełniania tego warunku. W kolejnym kroku odejmujemy od liczby 4248, która jest liczbą połączeń najbardziej ruchliwego przystanku, liczbę możliwych połączeń rozpatrywanego sąsiada. Węzły z większą liczbą połączeń mogą oferować więcej opcji dalszego przejścia i potencjalnie szybsze osiągnięcie celu. Tym samym przystanki z większą liczbą połączeń będą mili mniszą liczbę punktów karnych. Do znalezienia najbardziej ruchliwego przystanku stworzono prostą funkcję:

```
def the_busiest_stops(graph):
```

Stop	Number of connections
DWORZEC GŁÓWNY	4248
PL. GRUNWALDZKI	4067
GALERIA DOMINIKAŃSKA	4039
PL. JANA PAWŁA II	3579
DWORZEC AUTOBUSOWY	3514

1.5 Porównianie wyników

Do demontracji działania sprawdzono wyniki dla powyższych algorytmów.

```
Function dijkstra Took 0.0914 seconds
Starting time:  10:00:00
Total cost: 00:26:00
-------------------------------------------------------------------------------------------------------------
Line      Start                                          End
-------------------------------------------------------------------------------------------------------------
122       10:00:00  Kwiska                               10:03:00  Niedźwiedzia
3         10:03:00  Niedźwiedzia                         10:18:00  GALERIA DOMINIKAŃSKA
13        10:19:00  GALERIA DOMINIKAŃSKA                 10:22:00  Urząd Wojewódzki (Impart)
149       10:22:00  Urząd Wojewódzki (Impart)            10:26:00  PL. GRUNWALDZKI
-------------------------------------------------------------------------------------------------------------
```

```
Function a_star_inner_time Took 0.0166 seconds
Starting time:  10:00:00
Total cost for geo_distance heuristic: 00:26:00
-------------------------------------------------------------------------------------------------------------
Line      Start                                          End
-------------------------------------------------------------------------------------------------------------
122       10:00:00  Kwiska                               10:12:00  PL. JANA PAWŁA II
13        10:12:00  PL. JANA PAWŁA II                    10:22:00  Urząd Wojewódzki (Impart)
149       10:22:00  Urząd Wojewódzki (Impart)            10:23:00  most Grunwaldzki
13        10:24:00  most Grunwaldzki                     10:26:00  PL. GRUNWALDZKI
-------------------------------------------------------------------------------------------------------------
```

```
Function a_star_inner_line Took 0.0103 seconds
Starting time:  10:00:00
Total cost for line_change_heuristic heuristic: 00:29:00
-------------------------------------------------------------------------------------------------------------
Line      Start                                          End
-------------------------------------------------------------------------------------------------------------
12        10:04:00  Kwiska                               10:29:00  PL. GRUNWALDZKI
-------------------------------------------------------------------------------------------------------------
```

Algorytm A* optymalizujący liczbę przesiadek wykazał wynik zgodny z naszymi oczekiwaniami - uzyskaliśmy bezpośrednie połączenie z przystankiem docelowym. Algorytm Dijkstry i A* optymalizujące czas wykazują się dużą ilością przesiadek, natomiast czas takiego połączenie rzeczywiście jest mniejszy. Porównując te algorytmy między sobą możemy wnioskować, że A* jest zdecydowanie szybszy niż Dijkstra.

1.6 A*: modyfikacja

Udoskonalenie funkcji heurystycznej tak, aby lepiej odzwierciedlała rzeczywisty koszt dotarcia do celu, może skutecznie zmniejszyć liczbę węzłów rozpatrywanych przez algorytm, co z kolei może zmniejszyć czas obliczeń. Wybrano 4 heurystyki do porównywania działania algorytmu A*(kryterium czasu). Sprawdzono dla formuły Vincenty, Haversine, Manhattan, Euclidean. Ponieważ odległość Haversine'a i odległość geodezyjna są metodami przeznaczonymi do obliczania odległości na powierzchni sferycznej a odległość Manhattan i odległość euklidesowa są metodami obliczania odległości w prostym, płaskim układzie współrzędnych, wprowadzono pewien współczynnik korygujący, aby dostosować wyniki odległości. Wylosowano 5000 połączeń do sprawdzenia działania powyższych funkcji i obliczono średni czas działania algorytmu oraz koszt czasu (w sekundach).

Heuristic	Average time	Average cost
manhattan_distance	0.00703178	6170.90464548
haversine_distance	0.00687644	6021.58335034
euclidean_distance	0.00808569	6078.23589326
geo_distance	0.03113657	6027.34952051

Biorąc pod uwagę otrzymane wyniki geo_distance jest najwolniejszą heurystyką pod względem czasu obliczeń, co może wynikać z większej złożoności obliczeniowej formuły Vincenty w porównaniu do pozostałych metod. haversine_distance jest najszybszy spośród wszystkich testowanych heurystyk, więc najlepiej optymalizuje czas wykonania. Również jest najlepszym pod względem kosztu, chociaż poprzednio wspomniana geo_distance jest bardzo blisko najlepszej wartości (w naszym przypadku optymalizujemy czas i chcemy znaleść najszybsze połączenie). Jeżeli priorytetem jest czas obliczeń oraz zmniejszenie kosztu, to haversine_distance jest najlepszym wyborem. Natomiast w zastosowaniach, gdzie dokładność obliczeń odległości na elipsoidzie jest bardziej krytyczna, preferowanym rozwiązaniem może być geo_distance - jest wolniejsza ale algorytm działania jest dokładniejszy w porównaniu do innych heurystyk.

2 Zadanie 2

2.1 Przeszukiwanie Tabu: bez ograniczeń

Algorytm Tabu Search jest metodą do rozwiązywania problemów optymalizacyjnych, która polega na iteracyjnym eksplorowaniu przestrzeni rozwiązań. W kolejnych iteracjach algorytm przeszukuje sąsiednie rozwiązania i wybiera najlepsze z nich, które następnie uznaje za aktualne rozwiązanie. Wszystkie zmiany są rejestrowane na tzw. liście Tabu, która zawiera zestaw operacji zabronionych w przyszłych iteracjach. Dzięki temu algorytmowi nie wracamy do wcześniej odkrytych, mniej optymalnych rozwiązań. W wyniku zwiększa się szansa na dotarcie do optymalnego rozwiązania i zmniejsza się ryzyko utknięcia w lokalnym minimum (w tej implementacji funkcją kosztu jest czas dotarcia do przystanku docelowego). Ogólnie rzecz biorąc powstaję przed nami znany problem Komiwojażera, gdzie musimy odwiedzić wszystkie punkty i wrócić do punktu startowego. Implementacja algorytmu zrobiona na podstawie algorytmu przedstawionego na stronie. Najpierw ustawiamy ograniczenia, a mianowicie całkowitą liczbę iteracji oraz

maksymalną liczbę iteracji, które nie prowadzą do lepszych wyników.

```
max_iterations = n_stops ** 2
improve_limit = math.floor(2 * (math.sqrt(max_iterations)))
```

Na samym początku nasze rozwiązanie jest losowane: listę przystanków przypisujemy do zmiennej current_solution, a następnie losowo tworzymy początkową ścieżkę, która jest naszym pierwszym rozwiązaniem.

```
current_solution = stops
random.shuffle(current_solution)
```

Dalej wchodzimy do głównej pętli algorytmu, która będzie trwać do osiągnięcia maksymalnej liczby iteracji. Sprawdzamy czy ruch prowadzący do sąsiedniego rozwiązania nie znajduje się na liście tabu. Jeśli koszt sąsiedniego rozwiązania jest niższy niż koszt wcześniej rozpatrzonego w danej iteracji, aktualizujemy dane oraz dodajemy nasz ruch do listy tabu. W przypadku otrzymania lepszego rozwiązania resetujemy licznik operacji bez poprawy rozwiązania.

2.2 Przeszukiwanie Tabu: ograniczenie T

Długość tablicy tabu ma wpływ na liczbę ruchów blokowanych w kolejnych iteracjach algorytmu. Dłuższa lista tabu może pomóc w uniknięciu minimum lokalnego, ale może też spowodować zbyt szybkie zablokowanie ruchów, które są potrzebne do znalezienia najlepszego rozwiązania. W rozwiązaniu długość listy tabu została zdefiniowana jako liczba przystanków na odwiedzania. W przypadku przekroczenia limitu listy usuwamy najstarszy ruch, zapewniając tym samym jej stały rozmiar.

```
if len(tabu_list) > tabu_list_length:
    tabu_list.pop(0)
```

2.3 Przeszukiwanie Tabu: kryterium aspiracji

Mechanizm aspiracji w algorytmie Tabu Search pozwala na ominięcie ograniczeń nawet jeśli dany ruch znajduje się na liście zakazów, mechanizm aspiracji umożliwia jego wykonanie, pod warunkiem, że przyniesie on poprawę. Działa to na zasadzie specjalnego kryterium akceptacji, w naszym przypadku sprawdzamy, czy koszt tego rozwiązania jest niższy niż kryterium aspiracji.

```
neighbor_cost < aspiration_criteria:
```

Wartość aspiracji jest dynamicznie ustalana jako połowa wartości aktualnie najlepszego rozwiązania, co pozwala na elastyczne dostosowanie się do zmieniających się warunków poszukiwania optymalnej ścieżki.

```
aspiration_criteria = best_solution_cost * 0.5
```

2.4 Przeszukiwanie Tabu: próbkowanie

Odpowiedni wybór strategii próbkowania sąsiedztwa może przyspieszyć znalezienie optymalnego rozwiązania, umożliwić więcej iteracji w krótszym czasie, a także pomóc uniknąć lokalnych minimów i zwiększyć szanse na odkrycie lepszych rozwiązań. Do optymalizacji wybrano algorytm 2-opt. Algorytm 2-opt nie służy do wyznaczania trasy, a jedynie do ulepszania jej i polega na usunięciu z cyklu dwóch krawędzi i

zastąpieniu ich innymi krawędziami tak, aby utworzyć inny cykl.

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

2.5 Podsumowanie

Podczas implementacji napotkano problemy dotyczące dobrania odpowiednich wartości do funkcji heurystycznych. Uzyskanie optymalnego rozwiązania wykorzystując wyłącznie algorytm A* okazało się skomplikowane bez łączenia go z innymi technikami przeszukiwania i posiadania szczegółowej wiedzy na temat linii.

3 Bibliografia

https://www.youtube.com/watch?v=tIDhFPhrCbU Tabu search introduction
https://www.geeksforgeeks.org/what-is-tabu-search/ What is tabu
https://en.wikipedia.org/wiki/Vincenty%27s_formulae formuła Vincenty
https://en.wikipedia.org/wiki/Haversine_formula formuła Haversine
https://en.wikipedia.org/wiki/Taxicab_geometry formuła Manhattan
https://en.wikipedia.org/wiki/Euclidean_distance formuła Euclidean
https://en.wikipedia.org/wiki/2-opt 2-opt
https://docs.python.org/3/library/heapq.html heapq
https://geopy.readthedocs.io/en/stable/#module-geopy.distance geopy
https://pypi.org/project/haversine/ haversine
https://pandas.pydata.org/ pandas