Analiza danych BDOT10k w bazie Neo4j

Autorzy: Julia Zapała, Lidia Teleszko, Izabella Kaim

1. Cel

Celem zadania było zapisanie danych BDOT10k w bazie Neo4j oraz uruchomienie na nich algorytmów do wyszukiwania najkrótszej ścieżki.

2. Wstęp teoretyczny

Graf jest złożoną strukturą, w której dane przedstawia się za pomocą wierzchołków oraz krawędzi łączących te wierzchołki. Jednym z rodzajów grafu jest graf skierowany, w którym krawędź z jednego wierzchołka do drugiego ma określony kierunek i tylko w tym kierunku może odbywać się ruch. Wyróżnia się również graf nieskierowany, w którym krawędź nie ma kierunku i ruch może odbywać się obustronne z punktu A do B i z B do A. Graf może posiadać wagi, to znaczy każda krawędź może mieć przypisany niezbędny koszt lub zasoby do przejścia przez daną krawędź. Kosztem może być np. czas dojazdu z jednego miasta do drugiego.

W ćwiczeniu wykorzystywane są dwa główne algorytmy grafowe: Algorytm Dijkstry i A*. Algorytm Dijkstry to jedno z narzędzi, które służy do znajdowania najkrótszej ścieżki między dwoma punktami w grafie o nieujemnych wagach krawędzi. Algorytm rozpoczyna swoje działanie od wyboru punktu startowego. Następnia analizuje wszystkie drogi, przypisując każdemu wierzchołkowi odpowiednią wartość wag: nieskończoność dla niedostępnych punktów oraz wag krawędzi dla sąsiadów punktu startowego. Następnie wybierany jest wierzchołek o najmniejszej wadze, oznaczany jako odwiedzony, a jego sąsiadom aktualizuje się wagi, jeśli nowa ścieżka okaże się krótsza. Proces jest powtarzany do momentu, aż wszystkie wierzchołki zostaną odwiedzone lub znajdzie się najkrótsza trasa do punktu końcowego.

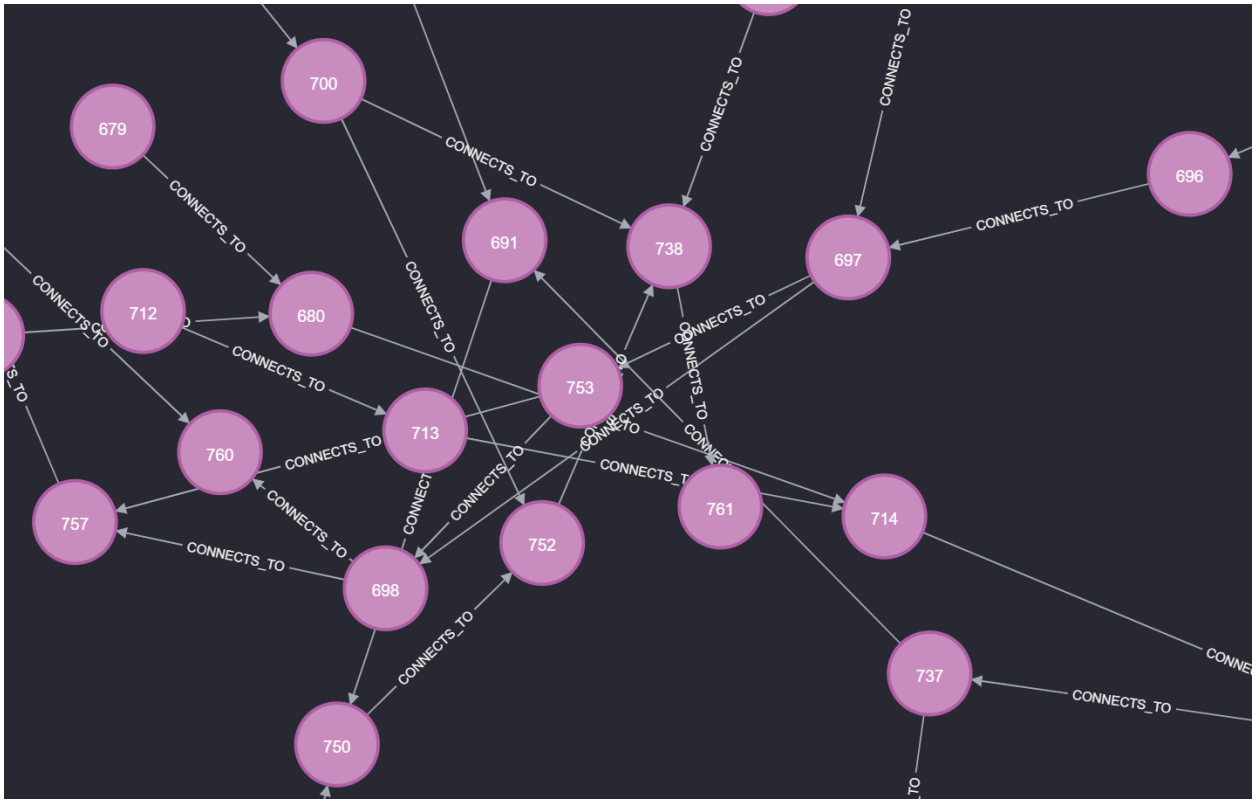
Algorytm A* jest rozszerzeniem algorytmu Dijkstry, który wykorzystuje heurystykę w celu efektywnego szukania najkrótszej ścieżki. Podobnie jak w algorytmie Dijkstry, na początku jest wybierany punkt początkowy, a następnie iteracyjnie sprawdzane są kolejne wierzchołki, uwzględniając zarówno koszt, jak i heurystykę, czyli koszt dojścia z danego wierzchołka do celu. Wierzchołki o najniższej łącznej wartości kosztu są odwiedzane priorytetowo, co przyspiesza proces. Zakończenie algorytmu następuje, kiedy zostanie znaleziona najkrótsza trasa.

Neo4j to popularna baza grafowa, która umożliwia przechowywanie i zarządzanie danymi w postaci grafów. Dzięki zastosowaniu tej struktury Neo4j jest szczególnie efektywna w rozwiązywaniu problemów związanych z analizą sieci. Baza grafowa pozwala na przechowywanie danych o wierzchołkach, relacjach oraz ich właściwości, co czyni ją idealnym narzędziem do pracy z grafami o różnej złożoności.

Biblioteka Neo4j Graph Data Science to zestaw narzędzi stworzony do zaawansowanej analizy grafów, który pozwala na efektywne uruchomienie algorytmów grafowych, takich jak algorytm Dijkstry czy A*. GDS umożliwia tworzenie grafów w pamięci, co przyspiesza obliczenia.

3. Realizacja zadania

Zadanie rozpoczęto od stworzenia bazy danych Neo4j. Następnie dodano do niej dane BDOT10k dla Torunia. Ten krok wykonano za pomocą skryptu napisanego w python. Za pomocą URL, hasła i nazwy użytkownika połączono się z bazą danych. Kod za pomocą geopandas wczytał plik .shp, odbyła się iteracja po każdym wierszu, wydobyto współrzędne początku i końca każdej drogi. Wykonano transformację z układu EPSG:2180 na EPSG:4326. Na podstawie współrzędnych stworzono węzły oraz relacje między nimi. Ostatnim krokiem był zapis danych do bazy.



Rys.1. Fragment powstałego grafu

Po dodaniu danych do bazy, można było przejść do projekcji grafu do pamięci. Podczas projekcji wzięto pod uwagę, aby graf nie był skierowany. Uwzględniono również właściwości wierzchołków i krawędzi (współrzędne i długości).

```
MATCH (source:Point)-[r:CONNECTS_TO]-(target:Point)
RETURN gds.graph.project(
  'myGraph',
  target,
  source,
  {
  sourceNodeProperties: source { .x, .y },
  targetNodeProperties: target { .x, .y },
  relationshipProperties: r{.length}
  }
)
```

Rys. 2. Projekcja grafu do pamięci

Ostatnim krokiem było uruchomienie funkcji Dijkstra. Numeracja wierzchołków w powstałym grafie zaczyna się od 0, a w oryginalnych danych od 1, dlatego aby zachować spójność danych, w funkcji uwzględniono różnice w podawanych wierzchołkach. Jako wynik otrzymano ścieżkę oraz koszt przejścia po tej ścieżce, czyli długość. Dla porównania wyników, dla tych samych węzłów uruchomiono algorytm Dijkstry z projektu 1.

```
CALL gds.shortestPath.dijkstra.stream('myGraph', {
  sourceNode: 774 - 1,
  targetNode: 195- 1,
  relationshipWeightProperty: 'length'
})
YIELD index, sourceNode, targetNode, totalCost, nodeIds, costs, path
WITH index, sourceNode, targetNode, totalCost, nodeIds, costs, path
RETURN
  index,
  sourceNode + 1 AS SourceNodeID,
  targetNode + 1 AS TargetNodeID,
  totalCost,
  [n IN nodeIds | n + 1] AS NodeIds,
  costs,
  [n IN nodes(path) | ID(n) + 1] AS path
ORDER BY index;
```

Rys. 3. Algorytm Dijsktry

index	SourceNodeID	TargetNodeID	totalCost	NodeIds
0	774	195	412.2407900334432	[774, 769, 194, 195]

Rys. 4. Wyniki dla biblioteki GDS

```
Shortest path dijkstra: 774 -> 769 -> 194 -> 195
Path length dijkstra: 412.2407900339673
```

Rys. 5. Wyniki dla algorytmu Dijkstry

Uzyskane wyniki pokrywają się, co potwierdza prawidłowość działania algorytmów.

Dla wczytanych danych uruchomiono również algorytm A*. Algorytm wykorzystuje length jako koszt, a heurystyka (odległość euklidesowa) jest liczona na podstawie współrzędnych. Aby zachować poprawność działania należało zadbać, aby współrzędne wczytanych danych były w układzie EPSG:4326 oraz aby graf był nieskierowany.

```
CALL gds.shortestPath.astar.stream('myGraph', {
  sourceNode: 774-1,
  targetNode: 195-1,
  latitudeProperty: 'x',
  longitudeProperty: 'y',
  relationshipWeightProperty: 'length'
})
YIELD index, sourceNode, targetNode, totalCost, nodeIds, costs, path
RETURN
  index,
  sourceNode + 1 AS SourceNodeID,
  targetNode + 1 AS TargetNodeID,
  totalCost,
  [n IN nodeIds | n + 1] AS NodeIds,
  costs,
  [n IN nodes(path) | ID(n) + 1] AS path
ORDER BY index;
```

Rys. 6. Algorytm A*

index	SourceNodeID	TargetNodeID	totalCost	NodeIds
0	774	195	412.2407900334432	[774, 769, 194, 195]

Rys. 7. Wyniki dla biblioteki GDS

```
Shortest path A*: 774 -> 769 -> 194 -> 195
Path length A*: 412.2407900339673
```

Rys. 8. Wyniki dla algorytmu A*

Tak samo jak dla algorytmu Dijkstry, algorytm ma bardzo zbliżone wyniki dla wywołań w Neo4j z użyciem biblioteki GDS oraz dla wywołań w python. Potwierdza to poprawność obu algorytmów.

4. Wnioski

1. Wykorzystanie Neo4j do przechowywania danych BDOT10k oraz ich analiza potwierdziło przydatność bazy grafowej do pracy z danymi przestrzennymi.
2. Przeprowadzone testy dla algorytmu Dijkstry oraz A* potwierdziły spójność implementacji w bibliotece GDS oraz w python. Wskazuje to na poprawność działania obydwu rozwiązań.
3. Algorytm Dijkstry i A*, pomimo że są innymi algorytmami, w niektórych wypadkach mogą dostarczać takich samych wyników. A* uwzględnia heurystykę, która pozwala na lepszą efektywność wyszukiwania ścieżki.
4. Rozwiązania oparte na Neo4j i algorytmach grafowych mogą być szeroko stosowane w analizie sieci transportowych, optymalizacji tras oraz modelowaniu danych przestrzennych.