Wojskowa Akademia Techniczna

Laboratorium

Systemy Baz Danych

Temat: Projekt i implementacja

Sieciowej Bazy Danych miast

inż. Patryk Bąk

Grupa: I7D2S4

Spis treści

Analiza problemu

Sieciowe bazy danych często wykorzystywane są do poszukiwania najkrótszej drogi. Znalezienie takiej trasy o najmniejszej długości, która pozwoli na odwiedzenie tylko raz każdego z miast i powrót do miasta początkowego nazywamy problemem komiwojażera (TSP) i jest on jednym z problemów milenijnych. Korzystając z systemu neo4j można bardzo szybko stworzyć bazę danych przeszukującą błyskawicznie wiele łuków i węzłów [1].

Założenia

1. Utworzona baza danych ma zawierać kilka przykładowych miast, reprezentowanych poprzez węzły i połączonych przy pomocy łuków,
2. Nazwy miast nie istnieją w rzeczywistym świecie (mogą jednak istnieć w rzeczywistości wirtualnej),
3. Każde miasto powinno posiadać etykietę *City* oraz właściwość *name* określającą nazwę miasta, *population* oznaczającą liczbę obywateli zamieszkującą to miasto, a także *capital*, która określa, czy dane miasto jest stolicą,
4. Właściwość *capital* dla miasta będącego stolicą powinna być unikalna,
5. Każde miasto ma posiadać własne wydarzenia (*event*), zabytki (*monument*), dworce/stocznie (*transport*) oraz uczelnie wyższe (*university*),
6. Miasta są połączone ze sobą w sposób taki, że tworzą cykl z jednym miastem centralnym, z którym połączone są wszystkie inne,
7. Łuki (relacje) łączące miasta powinny posiadać atrybut *distance*, który odpowiada odległości między miastami w kilometrach - odległości te uwzględniają utrudnienia związane z charakterystyką terenu, przez które prowadzi droga,
8. Odległości pomiędzy miastem A i miastem B oraz miastem B i miastem A powinny być różne,
9. Każdy węzeł oraz relacja muszą posiadać swój indeks.

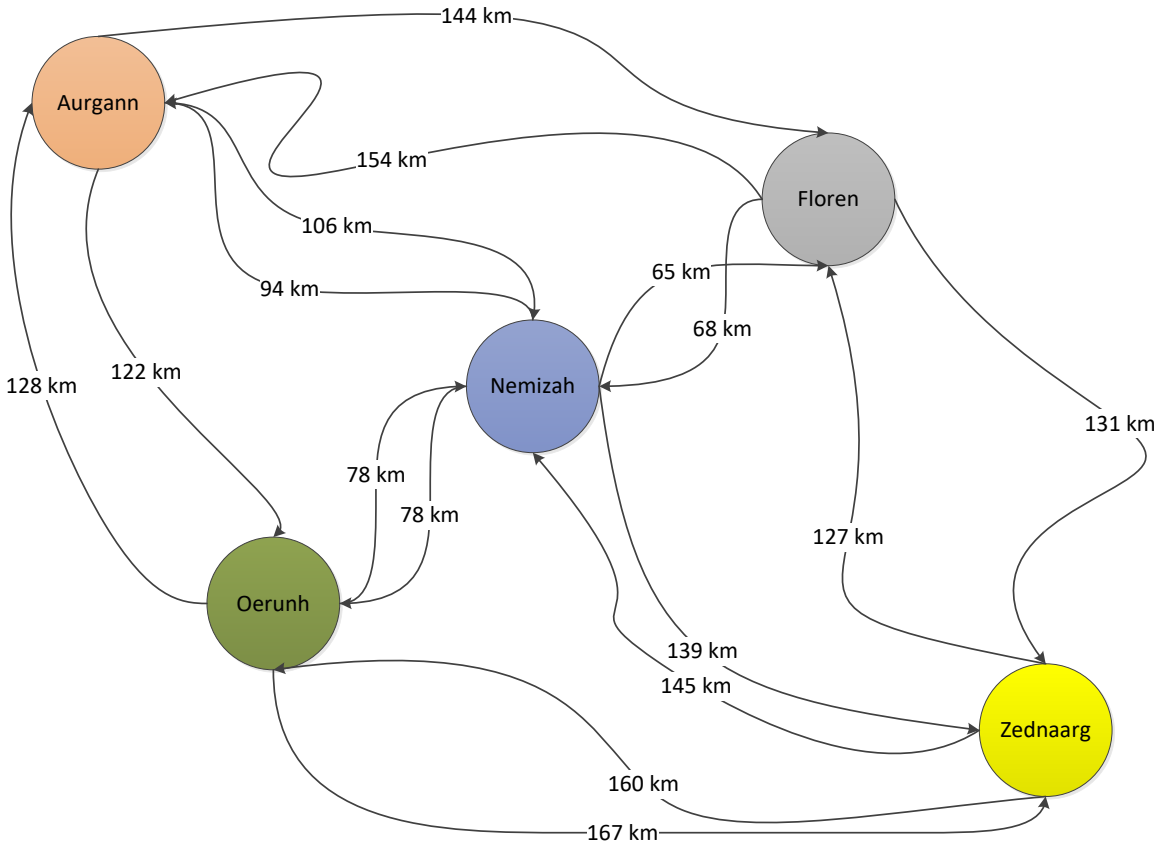
Przypadki użycia

Mając mapę miast z zadanymi odległościami w obu kierunkach można zlecić bazie danych znalezienie najkrótszej drogi z miasta A do miasta B, bądź z miasta A do miasta N, przechodzącą przez miasta B, C, …, M. Przy użyciu algorytmów z biblioteki Graph Algorithms jest to dosyć łatwe zadanie. Pozwala to na bardzo wydajne rozwiązywanie problemów logistycznych dla niewielkich ilości węzłów. Takie problemy są często rozwiązywane na potrzeby transportu. Rozwijając tą koncepcję, można dodać dodatkowe relacje unikalne dla

każdego miasta (szkoły, zabytki, lotniska) [2] i znaleźć odpowiednie miejsce przeszukując bazę (taki mechanizm stosowany jest np. w Mapach Google).

Projekt

Schematyczny układ mapy miast jest przedstawiony na diagramie poniżej:



Należy zauważyć, że aby istniał cykl przechodzący przez wszystkie miasta, połączenia między węzłami istnieją w obie strony (każde dwa węzły posiadają między sobą dwie relacje).

Tabela przedstawiająca węzły miast:

Węzły	Etykieta	Własności		
		name	*population*	*capital*
n1	City	Aurgann	7849	-
n2	City	Floren	12645	-

n3	City	Nemizah	34114	True
n4	City	Oerunh	8662	-
n5	City	Zednaarg	21045	-

gdzie: *name* – nazwa miasta, *population* – liczba obywateli, *capital* – czy miasto jest stolicą;

Tabela przedstawiająca węzły wydarzeń:

Węzły	Etykieta	Własności	
		name	*type*
n6	Event	Nemizah Festival	Concert
n7	Event	Night Of The Raven	Show

gdzie: *name* – nazwa wydarzenia, *type* – rodzaj wydarzenia;

Tabela przedstawiająca węzły zabytków:

Węzły	Etykieta	Własności				
		name	*built*	*age*	*style*	*area*
n8	Monument	Zednaarg Castle	1434	Middleages	Gothic	312,8
n9	Monument	Nemizah Palace	1688	Modern	Baroque	435,2
n10	Monument	Sultans Palace	1312	Middleages	Gothic	355,1
n11	Monument	Nemizah City Hall	1792	Modern	Gothic Revival	234,9
n12	Monument	Floren Castle	1121	Middleages	Roman	241,6
n13	Monument	Oerunh Therms	104	Antiquity	Roman	144,2
n14	Monument	Old Crane	1475	Middleages	Gothic	0,01

gdzie: *name* – nazwa zabytku, *built* – rok wybudowania, *age* – epoka, z której zabytek pochodzi, *style* – styl architektoniczny, *area* – obszar, jaki zabytek zajmuje, wyrażony w hektarach [ha];

Tabela przedstawiająca węzły dworców i stoczni:

Węzły	Etykieta	Własności						
		name	*employed*	*numer_of_ airplanes*	*numer_of_ trains*	*numer_of_ buses*	*numer_of_ ships*	*area*
n15	Transport	Nemizah Airport	2215	15	0	0	0	1845,4
n16	Transport	Nemizah Central Station	1156	0	20	30	0	732,8
n17	Transport	Floren Central Station	512	0	11	15	0	512,2
n18	Transport	Zednaarg Train Station	659	0	12	0	0	454,1
n19	Transport	Aurgann Central Station	815	0	14	10	0	554,1
n20	Transport	Aurgann Docks	241	0	0	0	32	781,2

gdzie: *name* – nazwa obiektu, *employed* – liczba zatrudnionych pracowników, *numer_of_airplanes* – liczba samolotów, *numer_of_trains* – liczba pociągów, *numer_of_buses* – liczba autobusów, *numer_of_ships* – liczba statków, *area* – obszar, jaki obiekt zajmuje, wyrażony w hektarach [ha];

Tabela przedstawiająca węzły uczelni wyższych:

Węzły	Etykieta	Własności			
		name	*employed*	*students*	*area*
n21	University	Nemizah Academy	4461	25324	781,2
n22	University	Nemizah University	4822	19376	655,8
n23	University	Floren School of Technology	1511	8125	485,1
n24	University	Zednaarg University	3521	9911	519,4

gdzie: *name* – nazwa uczelni, *employed* – liczba zatrudnionych pracowników, *students* – liczba studentów, *area* – obszar, jaki uczelnia zajmuje, wyrażony w hektarach [ha];

Tabela przedstawiająca relacje między miastami:

Łuki	Nazwa	Własności
		distance
(n1)->(n2)	path	144
(n2)->(n5)	path	131
(n5)->(n4)	path	160
(n4)->(n1)	path	128
(n3)->(n1)	path	94
(n3)->(n2)	path	65
(n3)->(n4)	path	78
(n3)->(n5)	path	139
(n2)->(n1)	path	154
(n5)->(n2)	path	127
(n4)->(n5)	path	167
(n1)->(n4)	path	122
(n1)->(n3)	path	106
(n2)->(n3)	path	68
(n4)->(n3)	path	78
(n5)->(n3)	path	145

gdzie: *distance* – odległość między miastami w kilometrach (w stronę zgodną ze zwrotem łuku);

Tabela przedstawiająca relacje między obiektami i miastami:

Łuki	Nazwa	Własności	
		date	*place*
(n6)->(n3)	event	20180621	Ervan Stadium
(n7)->(n2)	event	20180704	Betlaar Shopping Centre
(n8)->(n5)	monument	-	-
(n9)->(n3)	monument	-	-
(n10)->(n4)	monument	-	-
(n11)->(n3)	monument	-	-
(n12)->(n2)	monument	-	-
(n13)->(n4)	monument	-	-
(n14)->(n1)	monument	-	-
(n15)->(n3)	transport	-	-
(n16)->(n3)	transport	-	-
(n17)->(n2)	transport	-	-
(n18)->(n5)	transport	-	-
(n19)->(n1)	transport	-	-
(n20)->(n1)	transport	-	-

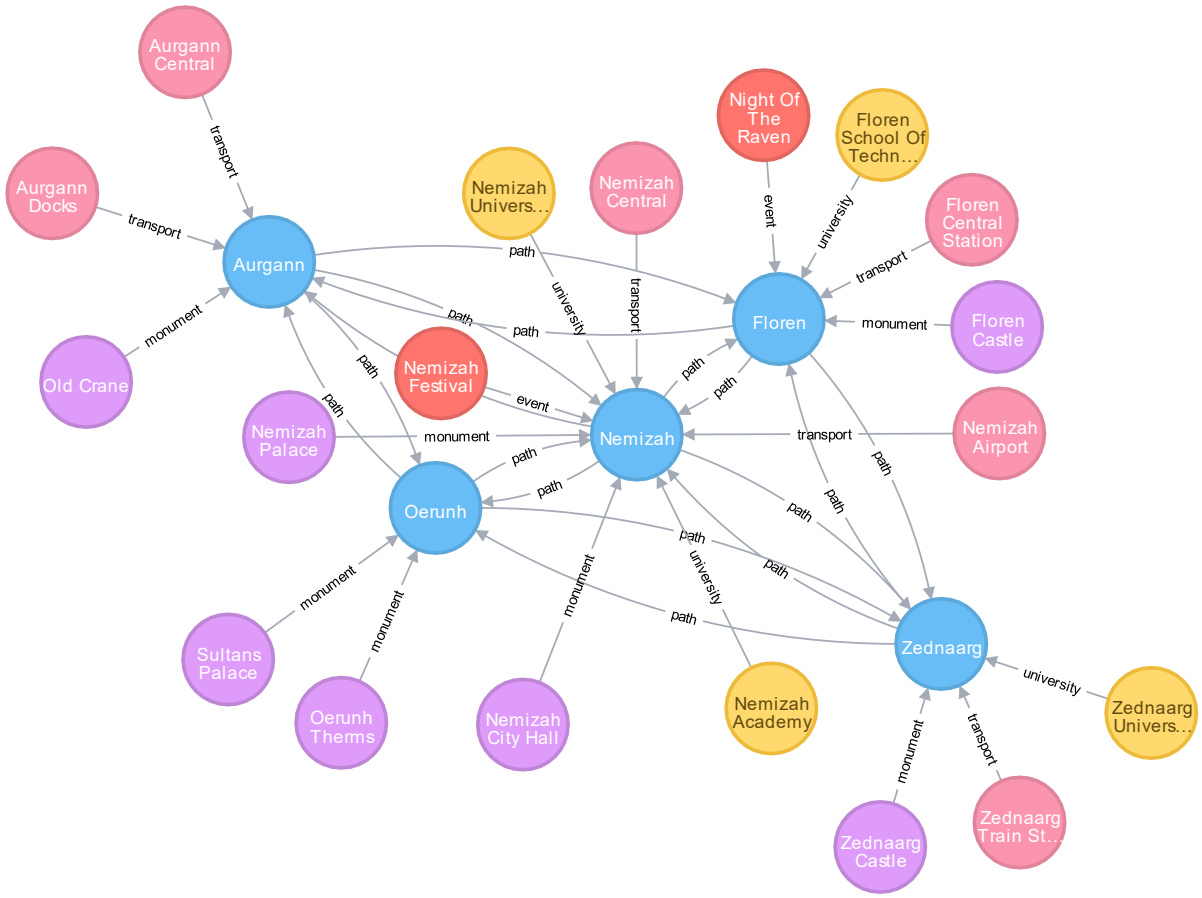
(n21)->(n3)	university	-	-
(n22)->(n3)	university	-	-
(n23)->(n2)	university	-	-
(n24)->(n5)	university	-	-

gdzie: *date* – data wydarzenia, wyrażona w formacie rrrrmmdd (bez separatorów), *place* – miejsce, gdzie odbędzie się wydarzenie (wewnątrz miasta);

Implementacja

Neo4j

Przy tworzeniu powyższej bazy danych starałem się podać wszystkie najistotniejsze dane dotyczące węzłów i łuków. Węzeł reprezentuje pojedyncze miasto i może być znaleziony poprzez zapytanie **MATCH…RETURN**, podobnie łuki opisują ścieżki między miastami oraz rodzaje obiektów w nich istniejące. Dzięki zastosowaniu relacji jest możliwe znalezienie najkrótszej ścieżki łączącej dwa miasta. Łatwo można odnaleźć miasto będące stolicą. Wystarczy użyć frazy *capital*, ponieważ węzeł reprezentujący to miasto ma unikalną własność *capital* równą *TRUE*. Po wykonaniu zapytań, baza danych wygląda następująco:



Poniżej znajduje się lista zapytań wraz z opisem:

```
CREATE CONSTRAINT ON (n3:City) ASSERT n3.capital IS UNIQUE;
```

Utworzona została unikalna wartość dla miasta „Nemizah”. Dzięki temu nie może być 2 stolic. Jest to pierwsze zapytanie, aby uniknąć błędów związanych z tworzeniem węzłów Następnie tworzę węzły będące miastami:

```
CREATE (n1:City {name: 'Aurgann', population: 7849}),
 (n2:City {name: 'Floren', population: 12645}),
 (n3:City {name: 'Nemizah', population: 34114, capital: TRUE}),
 (n4:City {name: 'Oerunh', population: 8662}),
 (n5:City {name: 'Zednaarg', population: 21045}),
```

Od teraz wartość *TRUE* własności *capital* będzie unikalna dla miasta „Nemizah”. Kolejnym krokiem jest utworzenie ścieżek pomiędzy miastami, wraz z atrybutem *distance*, którego wartość jest odległością prawdziwą przy poruszaniu się w wyznaczonym kierunku:

```
(n1)-[:path {distance: 144}]->(n2),
(n2)-[:path {distance: 131}]->(n5),
...
```

```
(n4)-[:path {distance: 78}]->(n3),

(n5)-[:path {distance: 145}]->(n3),
```

Następnie tworzę kolejne węzły. Będą one reprezentować wydarzenia. Dzięki temu możliwa jest zmiana jego daty bądź przypisanie go do innego miasta:

```
(n6:Event {name: 'Nemizah Festival', type: 'Concert'}),

(n7:Event {name: 'Night Of The Raven', type: 'Show'}),
```

Kolejne węzły określają obiekty, które istnieją w konkretnym mieście. Należą do nich zabytki (*Monument*), transport (*Transport*), uczelnie (*University*). Nazwy węzłów dla odróżnienia podaję zaczynając od wielkiej litery:

```
(n8:Monument {name: 'Zednaarg Castle', built: 1434, age: 'Middleages', style: 'Gothic', area: 312.8}),

...

(n14:Monument {name: 'Old Crane', built: 1475, age: 'Middleages', style: 'Gothic', area: 0.01}),


(n15:Transport {name: 'Nemizah Airport', employed: 2215, number_of_airplanes: 15, number_of_trains: 0, number_of_buses: 0, number_of_ships: 0, area: 1845.4}),

...

(n20:Transport {name: 'Aurgann Docks', employed: 241, number_of_airplanes: 0, number_of_trains: 0, number_of_buses: 0, number_of_ships: 32, area: 781.2}),


(n21:University {name: 'Nemizah Academy', employed: 4461, students: 25324, area: 781.2}),

...

(n24:University {name: 'Zednaarg University', employed: 3521, students: 9911, area: 519.4}),
```

Po utworzeniu węzłów pora na połączenie ich z miastami:

```
(n6)-[:event {date: 20180621,  place: 'Ervan Stadium'}]->(n3),

(n7)-[:event {date: 20180704, place: 'Betlaar Shopping Centre'}]->(n2),


(n8)-[:monument]->(n5),

...

(n24)-[:university]->(n5)
```

Na koniec tworzę indeksy, które pozwolą szybko wyszukać pożądany węzeł czy łuk, posługując się jego własnością. Pierwsze zapytanie dodaje wartość unikalną dla własności *name* dla węzłów reprezentujących miasta, dzięki czemu nie będzie można stworzyć miasta o istniejącej już nazwie:

```
CREATE CONSTRAINT ON (n:City) ASSERT n.name IS UNIQUE;

CREATE INDEX ON :path(distance);

CREATE INDEX ON :Event(name);
```

```
...
CREATE INDEX ON :University(name);
```

Warto też wykonać zapytanie, które wyświetli utworzone węzły i relacje:

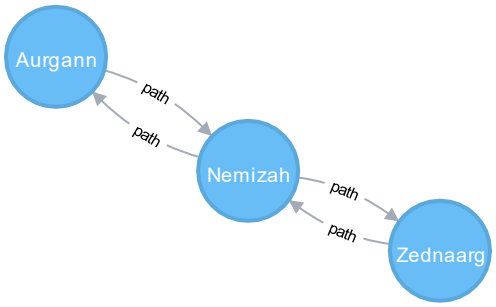
```
MATCH (n) RETURN n limit 100
```

Bazę w takim stanie można już testować.

1. Na początku znajduję odległość między dwoma węzłami:

```
MATCH p=(a {name: 'Aurgann'})-[r*2..5]->(b {name: 'Zednaarg'})
WITH p, relationships(p) AS rcoll
RETURN p, reduce(totalDistance=0, x IN rcoll| totalDistance + x.distance)
AS totalDistance
ORDER BY totalDistance
LIMIT 1
```

Znajduje ono takie *p*, które określa drogę z węzła początkowego *a* do węzła końcowego *b*. Zmienna *rcoll* zapisuje znalezioną ścieżkę *p* wraz z jej relacjami. W niej zmienna *x* przechowuje relacje, z których pobierana jest wartość własności *distance*. Zmienna *totalDistance* zwraca łączną odległość (dodaje wszystkie wartości *distance* łuków, przez które przechodzi droga). Następnie lista ścieżek jest sortowana względem wartości *totalDistance* od najmniejszej do największej (pierwsza droga jest najkrótsza), a *LIMIT 1* określa limit ścieżek jako 1, dzięki czemu zwracana jest najmniejsza droga. Po wykonaniu zapytania otrzymuję odległość między tymi dwoma miastami:



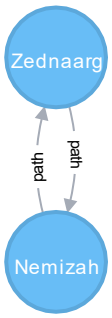
Zapytanie zwraca również odległości między poszczególnymi miastami oprócz łącznego dystansu między miastem Aurgann i Zednaarg:

"p"	"totalDistance"
[{"name":"Aurgann","capital":false,"population":7849},{**"distance"**:106},{"name":"Nemizah","capital":true,"population":34114},{"name":"Nemizah","capital":true,"population":34114},{**"distance"**:139},{"name":"Zednaarg","capital":false,"population":21045}]	**245**

2. Następnie postaram się znaleźć wszystkie węzły oddalone od ustalonego o określoną wartość. Niech będzie to wartość 140:

```
MATCH p=(a:City {name: 'Floren'})-[k:path]->(b:City)
WHERE k.distance <= 140
RETURN b
```

To zapytanie znajduje wszystkie miasta *b* przy założeniu, że ścieżka *p*, która biegnie od ustalonego miasta początkowego *a* (w tym przypadku miasta o nazwie „Floren”) do miasta końcowego *b* ma łączna wartość *distance* wszystkich łuków mniejszą lub równą 140. Wynik zapytania jest przedstawiony poniżej:



"b"
{"name":"Nemizah","capital":true,"population":34114}
{"name":"Zednaarg","population":21045}

Jak widać, zostały znalezione 2 miasta odległe od „Floren” o nie więcej niż 140 km.

3. Kolejne zapytanie będzie wykorzystywać słowo kluczowe FOREACH. Doda ono jeden dodatkowy atrybut o nazwie marked do każdego węzła na podanej ścieżce z początkową wartością TRUE:

```
MATCH p =(a:City)-[:path]->(b:City)
FOREACH (n IN nodes(p)| SET n.marked = TRUE )
```

Własność ta została dodana do każdego miasta. Można założyć, że oznacza ona miasta odwiedzone przez turystę (odwiedził wszystkie miasta). Wynik działania nie jest widoczny dla użytkownika, system informuje go jedynie o ilości wprowadzonych własności:

Set 32 properties, completed after 20 ms.

4. Następne zapytanie wykorzystuje słowo CONSTRAINT. Chcemy, aby stolicą było tylko jedno miasto, dlatego ustawiam własność *capital* na *TRUE* dla miasta „Nemizah”. Warto jednak uczynić to wartością unikalną po to, by nie dało się stworzyć drugiego węzła z własnością *capital* o wartości *TRUE*:

```
CREATE CONSTRAINT ON (City:Nemizah) ASSERT City.capital IS UNIQUE;
```

Warto tutaj zauważyć, że należy wykonać to zapytanie przed utworzeniem węzłów reprezentujących miasta. W przeciwnym razie narażamy się na błąd.

Dodam także zapytanie, które wykluczy możliwość utworzenia nowego węzła z etykietą *City* (miasta) z istniejącą już wartością własności *name* (nie ma dwóch miast o tej samej nazwie):

```
CREATE CONSTRAINT ON (n:City) ASSERT n.name IS UNIQUE;
```

Dla *n* węzłów z etykietą *City* (wszystkich) wartość własności *name* jest unikalna. Tutaj również program tylko informuje użytkownika o pomyślnym dodaniu własności:

Added 1 constraint, completed after 102 ms.

5. Ostatnim zapytaniem będzie znalezienie najkrótszej ścieżki z miasta „Oerunh” do miasta „Floren”. Użyję do tego funkcji *allShortestPaths* z biblioteki Graph Algorithms:

```
MATCH p = allShortestPaths((a {name: 'Oerunh'})-[r:path*..5]->(b { name:
'Floren'}))

  WITH p, relationships(p) AS rcoll

  RETURN p, reduce(totalDistance=0, x IN rcoll| totalDistance + x.distance)
AS totalDistance

  ORDER BY totalDistance

  LIMIT 1
```

Zapytanie działa analogicznie jak w punkcie 1, nie trzeba jednak filtrować wyniku. Zwraca ono najkrótszą ścieżkę wraz z odległościami od miast:

"p"	"totalDistance"
[{"marked":true,"name":"Oerunh","population":8662},{"distance":78},{"marked":true,"name":"Nemizah","capital":true,"population":34114},{"marked":true,"name":"Nemizah","capital":true,"population":34114},{"distance":65},{"marked":true,"name":"Floren","population":12645}]	143

Jak widać, najkrótsza trasa biegnie przez miasto „Nemizah” (143 km).

MS SQL 2017

Baza danych w MS SQL Server 2017 jest analogiczna, choć nieco się różni. Struktura to tabelki o nazwie etykiet węzłów i relacji z poprzedniej bazy danych, w których umieszczane są wszystkie o tej samej etykiecie:

```
CREATE TABLE dbo.Miasto(
MiastoID INT Identity(1,1),
nazwa VARCHAR NOT NULL,
populacja INT NOT NULL,
stolica BIT NOT NULL,
) AS NODE;


INSERT INTO dbo.Miasto(nazwa, populacja, stolica)
VALUES ('Aurgann', '7849', 0), ('Floren', '12645', 0), ('Nemizah', '34114', 1), ('Oerunh', '8662', 0), ('Zednaarg', '21045', 0);
```

Zdecydowałem się tutaj zmienić nazwy tabelek na polskie odpowiedniki z uwagi na fakt, że składnia języka SQL traktuje wiele z użytych wcześniej wyrazów jako słowa kluczowe, co powoduje konflikty. Drugi rodzaj tabelki, przedstawiający relacje *droga*, które odpowiadają łukom *path* z poprzedniej bazy:

```
CREATE TABLE droga(dystans INT) AS EDGE;


INSERT INTO droga($from_id, $to_id, dystans)
VALUES
((SELECT $node_id FROM Miasto WHERE MiastoID=1), (SELECT $node_id FROM Miasto WHERE MiastoID=2), 144),
...
((SELECT $node_id FROM Miasto WHERE MiastoID=5), (SELECT $node_id FROM Miasto WHERE MiastoID=3), 145);
```

Następnie tworzone są pozostałe węzły: *Wydarzenie*, *Zabytek*, *Transport*, *Uczelnia*, które odpowiadają kolejno węzłom z etykietami *Event*, *Monument*, *Transport*, *University*:

```
CREATE TABLE dbo.Wydarzenie(
WydarzenieID INT Identity(1,1),
nazwa VARCHAR NOT NULL,
typ VARCHAR NOT NULL,
```

```
) AS NODE;

INSERT INTO dbo.Wydarzenie(nazwa, typ)
VALUES ('Nemizah Festival', 'Concert'), ('Night Of The Raven', 'Show');

CREATE TABLE dbo.Zabytek(
ZabytekID INT Identity(1,1),
nazwa VARCHAR NOT NULL,
...
powierzchnia FLOAT NOT NULL,
) AS NODE;

INSERT INTO dbo.Zabytek(nazwa, rok_budowy, epoka, styl, powierzchnia)
VALUES
('Zednaarg Castle', 1434, 'Middleages', 'Gothic', 312.8),
...
('Old Crane', 1475, 'Middleages', 'Gothic', 0.01);

CREATE TABLE dbo.Transport(
TransportID INT Identity(1,1),
nazwa VARCHAR NOT NULL,
...
powierzchnia FLOAT NOT NULL,
) AS NODE;

INSERT INTO dbo.Transport(nazwa, liczba_pracownikow, liczba_samolotow,
liczba_pociagow, liczba_busow, liczba_statkow, powierzchnia )
VALUES
('Nemizah Airport', 2215, 15, 0, 0, 0, 1845.4),
...
('Aurgann Docks', 241, 0, 0, 0, 32, 781.2);
```

```
CREATE TABLE dbo.Uczelnia(
UczelniaID INT Identity(1,1),
nazwa VARCHAR NOT NULL,
...
powierzchnia FLOAT NOT NULL,
) AS NODE;


INSERT INTO dbo.Uczelnia(nazwa, liczba_pracownikow, liczba_studentow, powierzchnia)
VALUES
('Nemizah Academy', 4461, 25324, 781.2),
...
('Zednaarg University', 3521, 9911, 519.4);
```

Aby połączyć te węzły, tworzone są również łuki z etykietami właściwymi tym węzłom. Dodany został znacznik „_e”, aby odróżnić nazwy łuków od węzłów (język SQL nie jest *case sensitive*):

```
CREATE TABLE wydarzenie_e(data_wyd DATE, miejsce VARCHAR) AS EDGE;


INSERT INTO wydarzenie_e($from_id , $to_id, data_wyd, miejsce)
VALUES
((SELECT $node_id FROM Wydarzenie WHERE WydarzenieID=1), (SELECT $node_id FROM Miasto WHERE MiastoID=3), '06/21/2018', 'Ervan Stadium'),
((SELECT $node_id FROM Wydarzenie WHERE WydarzenieID=2), (SELECT $node_id FROM Miasto WHERE MiastoID=2), '07/04/2018', 'Betlaar Shopping Centre');


CREATE TABLE zabytek_e AS EDGE;
...
CREATE TABLE uczelnia_e AS EDGE;


INSERT INTO uczelnia_e($from_id , $to_id)
VALUES
((SELECT $node_id FROM Uczelnia WHERE UczelniaID=1), (SELECT $node_id FROM Miasto WHERE MiastoID=3)),
...
```

```
  ((SELECT $node_id FROM Uczelnia WHERE UczelniaID=4), (SELECT $node_id FROM
Miasto WHERE MiastoID=5));
```

Do utworzonych węzłów i relacji dodałem również niezbędne indeksy:

```
  CREATE UNIQUE INDEX stol ON Miasto(stolica);

  CREATE UNIQUE INDEX nazmiast ON Miasto(nazwa);

  CREATE INDEX odl ON droga(dystans);

  CREATE INDEX nazwyd ON Wydarzenie(nazwa);

  CREATE INDEX datawyd ON wydarzenie_e(data_wyd);

  CREATE INDEX miejscewyd ON wydarzenie_e(miejsce);

  CREATE INDEX nazzab ON Zabytek(nazwa);

  CREATE INDEX epokazab ON Zabytek(epoka);

  CREATE INDEX stylzab ON Zabytek(styl);

  CREATE INDEX naztrans ON Transport(nazwa);

  CREATE INDEX nazuni ON Uczelnia(nazwa);
```

Przejdźmy do zapytań. Pierwsze zapytanie tak jak poprzednio znajduje odległość między dwoma węzłami (podane zostały te same miasta):

```
SELECT a.nazwa, a.populacja, a.stolica, b.nazwa, b.populacja, b.stolica,
droga.dystans FROM Miasto a, droga,  Miasto b

WHERE a = Miasto.MiastoID(1) AND b = Miasto.MiastoID(5) AND MATCH(a -(r)->
b)

WITH p, relationships(p) AS rcoll

RETURN p, reduce(podroz=0, x IN rcoll| podroz + x.dystans) AS podroz

ORDER BY podroz

LIMIT 1
```

Następne zapytanie znajduje wszystkie węzły odległe od podanego o określoną wartość. Podobnie jak wcześniej, jest to wartość 140 (takie same dane):

```
SELECT a.nazwa, a.populacja, a.stolica, b.nazwa, b.populacja, b.stolica,
droga.dystans FROM Miasto a, droga,  Miasto b

WHERE droga.dystans <= 140 AND MATCH (a-(droga)->b)
```

Trzecie zapytanie dodaje kolumnę *odznacz* do tabelki *Miasto* dla wszystkich węzłów znajdujących się na ścieżce oraz ustawia wartość tej własności w każdym wierszu na 1:

```
SELECT a.nazwa, a.populacja, a.stolica, b.nazwa, b.populacja, b.stolica,
droga.dystans FROM Miasto a, droga,  Miasto b

WHERE MATCH (a-(droga)->b)
```

```
ALTER TABLE Miasto

ADD odznacz BIT;

SET Miasto.odznacz = 1 WHERE (a->b)
```

Wnioski

Sieciowe bazy danych są bardzo wydajnym narzędziem, pozwalającym na bardzo szybkie przeszukiwanie całej bazy oraz znajdowanie relacji pomiędzy węzłami. Muszę przyznać, że system Neo4j zainteresował mnie jako dosyć intuicyjne i atrakcyjne wizualnie narzędzie. Jest prosty w obsłudze, tworzenie bazy odbywa się w łatwy sposób i oferuje wiele możliwości. Niestety, występują tu również pewne szczegóły, które mogą wpłynąć na komfort użytkowania, jak na przykład ścisła kolejność wykonywania zapytań (węzły mogą utworzyć się dwukrotnie dla zaprezentowania związków między nimi). Mimo wszystko bardziej przypadł mi do gustu Neo4j, niż Microsoft SQL Server 2017. Ten drugi jest systemem przeznaczonym do tworzenia relacyjnych BD, bazującym na popularnym języku SQL. Umożliwia jednak tworzenie węzłów i łuków. Oczywiście, taki system daje dużo więcej możliwości, ale tworzenie zapytań o zawartość takiej bazy okazało się niestety mało intuicyjne i zmusiło mnie do rozwiązania wielu problemów. Dlatego uważam, że Neo4j jest zdecydowanie bardziej odpowiedni do takich zadań i realizuje je bardzo wydajnie.

Bibliografia

[1] https://neo4j.com/docs/developer-manual/3.4/.

[2] https://neo4j.com/graphgist/learning-cypher-with-san-francisco-bay-map.

Załączniki

1. Skrypt tekstowy generujący bazę danych w Neo4j
2. Pliki programu MS SQL 2017