SEARCH service

reference specification document

Version 1.0

TABLE OF CONTENTS

[Introduction 3](#_Toc499830399)

[Brief overview 4](#_Toc499830401)

[Service description 4](#_Toc499830402)

[Service architecture 4](#_Toc499830403)

[Service environment 4](#_Toc499830405)

[Rest api 5](#_Toc499830406)

[Input data format 6](#_Toc499830406)

[algorithm description 7](#_Toc499830406)

[base data model 9](#_Toc499830406)

[main service structure 9](#_Toc499830406)

[serviceassumptions 10](#_Toc499830406)

[FURTHER IMPROVEMENTS 10](#_Toc499830406)

**Introduction**

This document is Search service Specification, version 1.0. This specification contains all information required to understand design and architecture of developed service.

**Brief overview**

This section contains brief overview of the service and architecture design of the system.

**Service description**

Service has been built as a new feature which allows users of the app to determine the reachable towns/cities from a starting point in a given amount of time.

**Service architecture**

Search service was developed as a distributed, loosely coupled microservice. It employs lightweight communication protocol such as HTTP. Payload service data is represented via JSON.

**Service environment**

As a microservice, search service can fire up as an independent application. It will operate in \*nix environment. Actually, service has been developed to fire up in a cloud based systems.

**Rest API**

1. Find cities/tows that are reachable from a starting point in a given amount of time

|  |  |
| --- | --- |
| **POST http://{server}/search/city**  server – address of server  payload:  {  “cityName”:”Minsk”,  “duration”:”60”  }  cityName – name of city/town  duration – time of distance | |
| **Response example** | {  “cityName”:”duration”,  …  }  Example:  {  “Gomel”:”23”,  “Brest”:”34”  } |

**Payload data model:**

|  |  |
| --- | --- |
| **Field** | **Description** |
| String cityName | City/town name |
| Integer duration | duration |

**Input data format**

All nodes and edges are stored into in memory H2 database. Test data were populated into database using liquibase scripts. As application fires up, liquibase starts to create tables and populates data. The following tables describe data format:

GRAPH table:

|  |  |
| --- | --- |
| **Field** | **Description** |
| Id | Identifier of edge |
| from | code of node to start from |
| to | code of destination points(cities) |
| duration | Weight of edge |

CITY table:

|  |  |
| --- | --- |
| **Field** | **Description** |
| Id | Identifier of edge |
| code | code of node |
| name | Node name |

**Algorithm description**

Dijkstra's algorithm is an [algorithm](https://en.wikipedia.org/wiki/Algorithm) for finding the [shortest paths](https://en.wikipedia.org/wiki/Shortest_path_problem) between [nodes](https://en.wikipedia.org/wiki/Vertex_(graph_theory)) in a [graph](https://en.wikipedia.org/wiki/Graph_(abstract_data_type)), which may represent, for example, road networks. It was conceived by [computer scientist](https://en.wikipedia.org/wiki/Computer_scientist) [Edsger W. Dijkstra](https://en.wikipedia.org/wiki/Edsger_W._Dijkstra" \o "Edsger W. Dijkstra) in 1956 and published three years later.

The core idea of the Dijkstra algorithm is to continuously eliminate longer paths between the starting node and all possible destinations.

To keep track of the process, we need to have two distinct sets of nodes, settled and unsettled.

Settled nodes are the ones with a known minimum distance from the source. The unsettled nodes set gathers nodes that we can reach from the source, but we don’t know the minimum distance from the starting node.

Here’s a list of steps to follow in order to solve the SPP with Dijkstra:

* Set distance to *startNode* to zero.
* Set all other distances to an infinite value.
* We add the *startNode* to the unsettled nodes set.
* While the unsettled nodes set is not empty we:
  + Choose an evaluation node from the unsettled nodes set, the evaluation node should be the one with the lowest distance from the source.
  + Calculate new distances to direct neighbors by keeping the lowest distance at each evaluation.
  + Add neighbors that are not yet settled to the unsettled nodes set.

**private double**[] **distTo**; *// distTo[v] = distance of shortest s->v path***private** DirectedEdge[] **edgeTo**; *// edgeTo[v] = last edge on shortest s->v path***private** IndexMinPQ<Double> **pq**; *// priority queue of vertices***private int label**;  
**private** EdgeWeightedDigraph **graph**;

**public** DijkstraSP(EdgeWeightedDigraph G, **int** s, **int** label) {  
 **if**(*isNull*(G)){  
 **throw new** IllegalArgumentException(**"Graph can`t be null"**);  
 }  
 **this**.**graph** = G;  
 **this**.**label** = label;  
 **for** (DirectedEdge e : G.edges()) {  
 **if** (e.weight() < 0)  
 **throw new** IllegalArgumentException(**"edge "** + e + **" has negative weight"**);  
 }  
  
 **distTo** = **new double**[G.getV()];  
 **edgeTo** = **new** DirectedEdge[G.getV()];  
  
 validateVertex(s);  
  
 **for** (**int** v = 0; v < G.getV(); v++)  
 **distTo**[v] = Double.***POSITIVE\_INFINITY***;  
 **distTo**[s] = 0.0;  
  
 *// relax vertices in order of distance from s* **pq** = **new** IndexMinPQ<Double>(G.getV());  
 **pq**.insert(s, **distTo**[s]);  
 **while** (!**pq**.isEmpty()) {  
 **int** v = **pq**.delMin();  
 **for** (DirectedEdge e : G.adj(v))  
 relax(e);  
 }  
}  
  
*// relax edge e and update pq if changed***private void** relax(DirectedEdge e) {  
 **int** v = e.from(), w = e.to();  
 **if** ((**distTo**[w] > **distTo**[v] + e.weight()) && (**distTo**[v] + e.weight() <= **label**)) {  
 **distTo**[w] = **distTo**[v] + e.weight();  
 **edgeTo**[w] = e;  
 **if** (**pq**.contains(w)) **pq**.decreaseKey(w, **distTo**[w]);  
 **else pq**.insert(w, **distTo**[w]);  
 }  
}

**Base data model**

The application contain the following basic data model classes:

**DirectedEdge:**

|  |  |
| --- | --- |
| **Field** | **Description** |
| int v | Node position in array that from edge starts |
| int w; | Node position in array that to edge ends |
| int weight; | Weight of edge |

**EdgeWeightedDigraph:**

|  |  |
| --- | --- |
| **Field** | **Description** |
| int v | Number of vertices in this digraph |
| int e; | Number of edges in this digraph |
| int[] indegree; | Array, which responsible to store path from point to point |
| Container<DirectedEdge>[] adj; | Adjacency list for vertex v |

**Main service structure**

**SearchService:**

|  |  |  |  |
| --- | --- | --- | --- |
| **Method name** | **Parameters** | **Return value** | **Description** |
| findAllCitiesByParamms | SearchParamms paramms – contains from city name and duration | Map<String, Double> - map contains from city name and min duration according to search data and start point | Method returns map (city name, duration) according search data and start node. |

**Service assumptions**

1. Data is saved into in memory database
2. All graph parameters are saved into memory as objects
3. Simple cache implementation is applied
4. Simple Dijkstra algorithm is applied.

**Further improvements**

1. Store data in the persistent database
2. Type of transport not considered (user can trip by bus, plane, train). Due to type of transport, we may need to have different weights of edge
3. Enhance cache handling
4. Develop algorithm, which reads data from database consistently. It can help avoid out of memory issues when working with billions of nodes
5. Try to use “Memory Mapped Files”.