Assignment 7

Object-Relational Database Programming

1. **In this problem, you can not use arrays**.

   Consider the relation schema `Graph(source int, target int)` representing the schema for storing a directed graph G as a set of edges.

   Recall that G is *connected* if for each pair of vertices (s,t) in G with $s \neq t$, there exists a path in G from s to t, i.e., if for each pair of vertices (s,t) in G with $s \neq t$, (s,t) is in the transitive closure of G.

   An *articulation vertex* of G is a vertex $\mathbf{v}$ of G such that removing the edges in G with source or target $\mathbf{v}$ results in a graph that is **not** connected. More formally, $\mathbf{v}$ is an articulation vertex of G, if the graph

   $$G - (\{(s,t)|(s,t) \in G \wedge (s = \mathbf{v} \vee t = \mathbf{v}\})\}$$

   is **not** connected.

   We say that G is *bi-connected* if G does not have any articulation vertices.

   Write a PostgresQL program `biConnected()` that returns true if G is bi-connected and false otherwise.

2. **In this problem, you can not use arrays.**

   Implement the HITS authority-hubs algorithm which is a varaiant of a page ranking algorithm. For more information about the HITS algorithm consult

   `https://en.wikipedia.org/wiki/HITS_algorithm`
   `https://www.youtube.com/watch?v=jr3YGgfDY_E`

   The input data is given in a relation `Graph(source integer, target integer)` which represent the graph on which the HITS Algorithm operates.[1] So each vertex in this graph will receive an authority and a hub score.

   An important detail of the HITS algorithm concerns the normalization of the authority vector (analogously, the hub vector). This vector needs to be normalized to have norm = 1 after each iteration step. Otherwise, the algorithm will not converge.

   Normalization of a vector of numbers can be done as follows: If $\mathbf{x} = (x_1, \ldots, x_n)$ is a vector of real numbers, then its norm $|\mathbf{x}|$ is given by the formula $\sqrt{x_1^2 + \cdots + x_n^2}$. Therefore, you can normalize the vector $(x_1, \ldots, x_n)$ by transforming it to the vector $\frac{\mathbf{x}}{|\mathbf{x}|} = (\frac{x_1}{|\mathbf{x}|} + \cdots + \frac{x_n}{|\mathbf{x}|})$. The norm of this vector will be 1.

[1] This graph is called the *base* in the Wikipedia article.

3. **In this problem, you can use arrays, but only as a mechanism to represents sets of words**.

   Consider the relation schema `document(doc int, words text[])` representing a relation of pairs (d, W) where d is a unique document id and W denotes the set of words that occur in d.

   Let $\mathbf{W}$ denote the set of all words that occur in the documents and let t be a positive integer denoting a *threshold*.

   Let $X \subseteq \mathbf{W}$. We say that X is t-frequent if

   $$\texttt{count}(\{d|(d, W) \in \texttt{document}\,\text{and}\, X \subseteq W\}) \geq t$$

   In other words, X is t-*frequent* if there are at least t documents that contain all the words in X.

   Write a PostgreSQL program `frequentSets(t int)` that returns the relation of all t-frequent sets.

   In a good solution for this problem, you should use the following rule: if X is not t-frequent then any set Y such that $X \subseteq Y$ is not t-frequent either. In the literature, this is called the *Apriori* rule of the frequent itemset mining problem. This rule allows you to avoid examing supersets of sets that are not frequent. This can drastically reduce the search space.

4. **In this problem you can not use arrays**.

   Suppose you have a weighted undirected graph G stored in a ternary table with schema

   `Graph(source int, target int, weight int)`

   A triple (s, t, w) in Graph indicates that *Graph* has an edge (s, t) whose edge weight is w. (In this problem, we will assume that each edge weight is a positive integer.)

   Since the graph is undirected, whenever there is an weighted edge (s, t, w) in G, then (t, s, w) is also a weighted edge in the G. Below is an example of a graph G.

`Graph` G

source	target	weight
0	1	2
1	0	2
0	4	10
4	0	10
1	3	3
3	1	3
1	4	7
4	1	7
2	3	4
3	2	4
3	4	5
4	3	5
4	2	6
2	4	6

Implement Dijkstra's Algorithm as a PostgreSQL function `Dijkstra(s integer)` to compute the shortest path lengths (i.e., the distances) from some input vertex s in G to all other vertices in G. `Dijkstra(s integer)` should accept an argument s, the source vertex, and outputs a table `Paths` which represents the pairs (t, d) where d is the shortest distance from s to t. To test your procedure, you can use the graph shown above.

You can find a description of Dijkstra's algorithm as the following webpage `https://en.wikipedia.org/wiki/Dijkstra%27s_algorithm#Pseudocode`

When you apply Dijkstra(0), you should obtain the following `Paths` table:

target	distanceToTarget
0	0
1	2
2	9
3	5
4	9

5. **In this problem, you can not use arrays**.

   Consider the relation schema `Graph(source int, target int)` representing the schema for storing a directed graph G of edges.

   Let ‘red’, ‘green’, and ‘blue’ be 3 colors. We say that G is *3-colorable* if it is possible to assign to each vertex of G one of these 3 colors provided that, for each edge (s, t) in G, the color assigned to s is different than the color assigned to t.

   Write a PostgresQL program `threeColorable()` that returns true if G is 3-colorable, and false otherwise.

   (Hint: use a backtracking algorithm that finds a 3-color assignment to the vertices of G if such an assignment exists.)