Implement Connected Components Algorithms for pgRouting by the Boost Graph Library

1. Contact Details

Name:	Maoguang Wang
Nickname:	mg
Country:	China
Email:	xjtumg1007@gmail.com
Phone:	+86 18992859919
Github:	https://github.com/xjtumg
Personal blog:	www.xjtumg.me
Twitter:	XJTUmg

2. Title

Implement Connected Components Algorithms for pgRouting by the Boost Graph Library.

3. Synopsis

Connected components algorithms are used to analyze graph and solve problems (like 2-satisfiability problem[1]). There are three parts of connected components algorithms in the Boost Graph Library (BGL),

1. Connected components algorithm.
2. Strongly connected components algorithm.
3. Biconnected components algorithm.

I am proposing to add those BGL functionalities to pgRouting during this GSoC period.

4. Benefits to Community

Connected Components Algorithms, as an important part of graph theory, is widely applied in the field of graph analysis and graph contraction. To improve computation and analytical ability for graph in pgRouting, I choose to implement the algorithms in the BGL for pgRouting, and then, time permitting, implement more applications. Furthermore, Connected Components Algorithms can be used combining with other route planning algorithms to figure out problems (like following application 3) with greater efficiency. Consequently, all of these are going to help further development and applications in pgRouting.

Some possible applications:

1. Suppose you have a huge graph and you want to find how one vertex is related to another. Using connected components algorithms can solve it in linear time complexity.
2. 2-satisfiability problem. Boolean satisfiability determines whether we can give values (`True` or `False` only) to each boolean variable leading the value of the formula become `True` or not. If we can do so, we call formula is satisfiable, otherwise we call it unsatisfiable. 2-satisfiability is a special case of boolean satisfiability. Read more. Sample problem.

[1] 2-satisfiability - Wikipedia.

3. If you travel from city-to-city and you want some conditions (prices, weather, etc.) to hold true while picking airports, connected components algorithms could help you.
4. Finding bridges[2] in undirected graph. A bridge is an edge of the given graph whose deletion increases its number of connected components. Tarjan's algorithm based on DFS[3] can solve it in linear time complexity.

5. Deliverables

The deliverables would be:

1. Implementation of connected components algorithm for pgRouting by the BGL.
2. Implementation of strongly connected components algorithm for pgRouting by the BGL.
3. Implementation of biconnected components algorithm for pgRouting by the BGL.
4. Documentation and tests for the above-mentioned components.

6. Related Work

GraphStream,
igraph,
GSoC 2016: Flow Algorithms (Implemented flow algorithms for pgRouting by the BGL),
the Boost Graph Library 1.46.0.

7. Biographical Information

I am second year Information and Computational Science undergraduate from Xi'an Jiaotong University (XJTU) in Xi'an, Shaanxi, China. Algorithms and coding are my strengths and future direction. I am looking forward to pursue a career in the field of algorithms. I have much interest in open source development as it gets closest to what users want and it make developers free to create or improve programs which contribute to the real world.
I started writing code and delving into algorithms when I was in middle school. I am familiar with most of the algorithms in the BGL because of my abundant algorithm competition experience. I have not written any software or libraries, though I have received several acknowledgements in algorithm competitions. I want to challenge myself in the open source world and make more and more contributions. So I asked pgRouting team for help. Their feedback and guidance helped me a lot, and then I made my first contribution to pgRouting. I do know it is easier than what I propose to do in the next months, but I am convinced this is a good start, and I will look forward to finishing my GSoC smoothly.

8. Timeline

Community Bonding Period (May 4 to May 29, 2017):

1. Read the BGL docs.
2. Make a wiki page (TODO lists and weekly reports).

---

[2] Bridge (graph theory) - Wikipedia.

[3] Tarjan R. Depth-first search and linear graph algorithms[J]. SIAM journal on computing, 1972, 1(2): 146-160.

3. Develop a better understanding of postGIS, postgreSQL and PL/pgsql. Get familiar with postgreSQL procedural language.
4. Learn PgTAP for automated tests.
5. Get familiar with pgRouting architecture.
6. Review C++ videos and write a report.

Official Coding Period (May 30 to August 21, 2017):

Official Coding Period Phase 1 (May 30 to June 26, 2017):

Week 1 (May 30 to June 5, 2017):

1. Learn how to create documentation and tests.
2. Create initial documentation.
3. Analyze whether the current sample data is good for testing and documenting the functions. If not, create new sample data for these functions.

Week 2 to 3 (June 6 to June 19, 2017):

1. Learn how to implement functions for pgRouting by the BGL. Go through related work for a better understanding and skill on the implementation.

Week 4 (June 20 to June 26, 2017):

1. Prepare basic code and test framework for the upcoming implementations.

First evaluation period (June 26 to June 30, 2017):

1. Mentors evaluate me and I evaluate mentors of officail coding period phase 1.
2. Deliver initial documentation, basic code and test framework for the upcoming implementations.
3. This phase of learning would help a lot to the implementations.

Official Coding Period Phase 2 (June 27 to July 24, 2017):

Week 5 to 6 (June 27 to July 10, 2017):

1. Implement `pgr_connectedComponents()` function.
2. Create documentation for the first implementation.
3. Create tests for the first implementation.

Week 7 to 8 (July 11 to July 24, 2017):

1. Implement `pgr_strongComponents()` function.
2. Create documentation for the second implementation.
3. Create tests for the second implementation.

Second evaluation period (July 24 to July 28, 2017):

1. Mentors evaluate me and I evaluate mentors of officail coding period phase 2.
2. Deliver the first and the second implementation as above-mentioned.

Official Coding Period Phase 3 (July 25 to August 21, 2017):

Week 9 to 10 (July 25 to August 7, 2017):

1. Implement `pgr_biconnectedComponents()` function.
2. Create documentation for the third implementation.
3. Create tests for the third implementation.

Week 11 to 12 (August 8 to August 21, 2017):

1. Fix bugs and documentation details.
2. Prepare for final delivery.

Final evaluation period (August 21 to August 29, 2017):

1. Wrap up my projects and submit final evaluation of my mentors.

Weekly reports:
I will post a report to the soc@osgeo and the developer mailing list of my project weekly. And that at least answers the following questions:

1. What did you get done this week?
2. What do you plan on doing next week?
3. Are you blocked on anything?

Do you understand this is a serious commitment, equivalent to a full-time paid summer internship or summer job?
Yes, I completely understand. I am fully prepared for the work and will spend at least 40 hours a week. I have tremendous passion and power to learn and develop. Also, I believe, I am a self-motivated person as you need. If time permits, I would try to make at least one application that makes use of a connected component function. And I would love to keep contributing to pgRouting after this GSoC period.

Do you have any known time conflicts during the official coding period?
I do not have any conflict during the official coding period.

9. Studies

What is your school and degree?

School: Xi'an Jiaotong University(XJTU) in China.
Degree: Second year undergraduate in Information and Computational Science.

Would your application contribute to your ongoing studies/degree? If so, how?

Definitely yes! I have been enjoying coding and delving into algorithms since I was in middle school. For the study of algorithms, I received several acknowledgements in algorithm competitions, but I really want to got more breakthroughs on my computer skills. I am convinced the experience of GSoC in pgRouting which contains planning, developing, coding and writing documentation and tests will contribute much to my studies.

10. Programming and GIS

Computing experience

Operating systems:
Ubuntu 16.04.2 LTS, OSX 10.12, Windows 10.
Languages:
C, C++, Pascal, Python, Java, Perl, Bash.

Programming Contest:
Silver Medal, 59th, National Olympiad in Informatics(NOI China), 2013 Jul.
Gold Medal, 17th, ACM International Collegiate Programming Contest(ACM-ICPC) China-Final, 2016 Dec.

GIS experience:

I use various GIS related softwares and libraries like pgRouting, QGIS, osm2pgrouting.

GIS and pgRouting programming:

I have fixed all warnings in `pgrouting/src/trsp/src/GraphDefinition.cpp`, and I submit two pull requests as a part of 2.4 release.

11. GSoC participation

I have not participated to GSoC before and I didn't submit a proposal to any other organization.

12. Proposal: Connected Components Algorithms

Introduction

Connected components algorithms are important parts of graph algorithms. For this proposal, I am aiming to implement connected components algorithms for pgRouting by the BGL. For details,

1. **Connected components**[4]: The `connected_components()` functions in the BGL compute the connected components of an undirected graph. A connected components of an undirected graph is a set of vertices that all reachable from each other. The time complexity for the connected components algorithm using DFS-based approach is `O(V + E)`.
2. **Strongly connected components**[5]: The `strong_components()` functions in the BGL compute the strongly connected components of a directed graph. Finding strongly connected components is used to solve 2-satisfiability (2-SAT) problems. The time complexity for the strongly components algorithm using Tarjan's algorithm based on DFS is `O(V + E)`.
3. **Biconnected components**[6]: The `biconnected_components()` functions in the BGL compute the biconnected components of an undirected graph. The biconnected components of a graph are the maximal subsets of vertices such that removal of a vertex from a particular component will not disconnect the component. The time complexity for the biconnected components algorithm using Tarjan's algorithm based on DFS is also `O(V + E)`.

[4] the Boost Graph Library: Connected Components - 1.46.0.
[5] the Boost Graph Library: Strongly Connected Components - 1.46.0.
[6] the Boost Graph Library: Biconnected Components - 1.46.0.

Versions

Ubuntu 16.04.2 LTS
gcc/g++ (Ubuntu/Linaro 4.6.4-6ubuntu2) 4.6.4
cmake 3.5.1
Boost 1.46.0
pgRouting 2.4.0
PostgreSQL 9.5.6
PostGIS 2.2.1

Attention

To give an idea of the proposed functionality, the following code and examples does not necessarily reflect the final signatures or the final results column names.
Following C++ code are stored in my github repository.

Description of the edges_sql query for the following proposed functions[7]

edges_sql: an SQL query, which should return a set of rows with the following columns:

Column	Type	Default	Description
id	ANY-INTEGER		Identifier of the edge.
source	ANY-INTEGER		Identifier of the first end point vertex of the edge.
target	ANY-INTEGER		Identifier of the second end point vertex of the edge.
cost	ANY-NUMERICAL		Weight of the edge (source, target) ● When negative: edge (source, target) does not exist, therefore it's not part of the graph.
reverse_cost	ANY-NUMERICAL	-1	Weight of the edge (target, source), ● When negative: edge (target, source) does not exist, therefore it's not part of the graph.

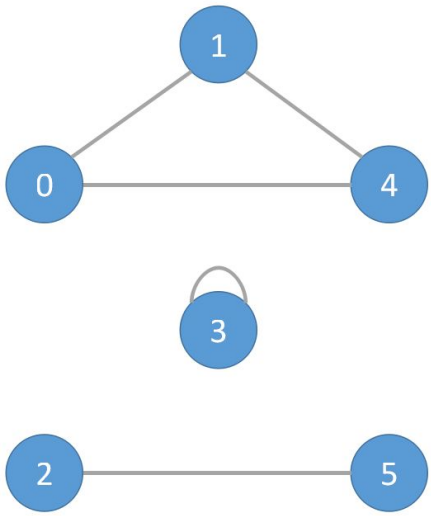
Where:
ANY-INTEGER: SMALLINT, INTEGER, BIGINT
ANY-NUMERICAL: SMALLINT, INTEGER, BIGINT, REAL, FLOAT

[7] similiar to description of the edges_sql query for dijkstra like functions
http://docs.pgrouting.org/2.4/en/pgRouting-concepts.html#description-of-the-edges-sql-query-for-dijkstra-like-functions

Connected components

Example

Figure 1: example of connected components.



Demo slides

https://docs.google.com/presentation/d/1MVKQR8DkmqhwaP57WSFStvU_kGuYzVUUcJPrhPjNRqg/present#slide=id.g1ee22e81a9_0_58

Use the BGL

Code

https://github.com/XJTUmg/ExampleCode/blob/master/ConnectedComponents/ConnectedComponents.cpp

Output

```
Total number of components: 3
Vertex 0 is in component 0
Vertex 1 is in component 0
Vertex 2 is in component 1
Vertex 3 is in component 2
Vertex 4 is in component 0
Vertex 5 is in component 1
```

Proposed function for pgRouting

Initial setup

```
CREATE TABLE edge_table (
      id BIGSERIAL,
      source BIGINT,
      target BIGINT,
      cost FLOAT,
```

```
      reverse_cost FLOAT
);

INSERT INTO edge_table (
      source, target,
      cost, reverse_cost) VALUES
(0, 1, 1, 1),
(1, 4, 1, 1),
(0, 4, 1, 1),
(2, 5, 1, 1),
(3, 3, 1, 1);
```

Procedure

I am proposing add `pgr_connectedComponents()` function to pgRouting.

```
SELECT * FROM pgr_connectedComponents (
      'SELECT id, source, target, cost, reverse_cost FROM edge_table');
```

Results

```
 seq | node_seq | node | component
-----+----------+------+-----------
   1 |        1 |    0 |         1
   2 |        2 |    1 |         1
   3 |        3 |    4 |         1
   4 |        1 |    2 |         2
   5 |        2 |    5 |         2
   6 |        1 |    3 |         3
(6 rows)
```

About `pgr_labelGraph` function[8]

`pgr_labelGraph` locates and labels sub-networks within a network which are not topologically connected. It works on undirected graph.

1. The functionality of `pgr_connectedComponents` is very similar as `pgr_labelGraph`.
   Analyze the above example using `pgr_labelGraph`:

   ```
   CREATE TABLE edge_table (
          id BIGSERIAL, source BIGINT, target BIGINT);
   CREATE TABLE
   INSERT INTO edge_table (source, target) VALUES
          (0, 1), (1, 4), (0, 4), (2, 5), (3, 3);
   INSERT 0 5
   SET client_min_messages TO WARNING;
   SET
   SELECT pgr_labelGraph(
          'edge_table', 'id', 'source', 'target', 'subgraph');
   ```

[8] pgr_labelGraph - Proposed

```
 pgr_labelgraph
---------------
 OK
(1 row)
SELECT subgraph, count(*) FROM edge_table group by subgraph;
 subgraph | count
----------+-------
        1 |     3
        3 |     1
        2 |     1
(3 rows)
```

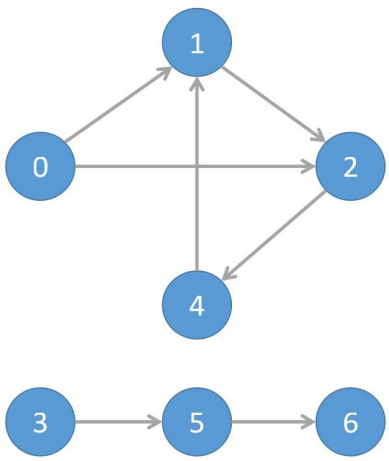
2. `pgr_labelGraph` uses BFS-based approach, the time complexity of this function is square level. But `connected_components` function in the BGL uses DFS-based approach with linear time complexity. In a word, `pgr_connectedComponents` will be much faster than `pgr_labelGraph`.
3. `pgr_connectedComponents` will return more columns which means provide more information to users. For each vertex, `pgr_connectedComponents` will allow users to find which component the vertex belongs.

To summarise, `pgr_labelGraph` can be replaced with `pgr_connectedComponents`.

Strongly connected components

Example

Figure 2: example of strongly connected components.



Demo slides

https://docs.google.com/presentation/d/1lbhAmmRm-a3IAZ3VahQOsA6SaH29ddOnFsoNTOHkGcE/present#slide=id.p

Use the BGL

Code

https://github.com/XJTUmg/ExampleCode/blob/master/StronglyConnectedComponents/StronglyConnectedComponents.cpp

Output

```
Total number of strongly connected components: 5
Vertex 0 is in strongly connected component 1
Vertex 1 is in strongly connected component 0
Vertex 2 is in strongly connected component 0
Vertex 3 is in strongly connected component 4
Vertex 4 is in strongly connected component 0
Vertex 5 is in strongly connected component 3
Vertex 6 is in strongly connected component 2
```

Proposed function for pgRouting

Initial setup

```
CREATE TABLE edge_table (
      id BIGSERIAL,
      source BIGINT,
      target BIGINT,
      cost FLOAT,
      reverse_cost FLOAT
);

INSERT INTO edge_table (
      source, target,
      cost, reverse_cost) VALUES
(0, 1, 1, -1),
(1, 2, 1, -1),
(0, 2, 1, -1),
(2, 4, 1, -1),
(1, 4, -1, 1),
(3, 5, 1, -1),
(5, 6, 1, -1);
```

Procedure

I am proposing add `pgr_strongComponents()` function to pgRouting.

```
SELECT * FROM pgr_strongComponents (
      'SELECT id, source, target, cost, reverse_cost FROM edge_table');
```

Results

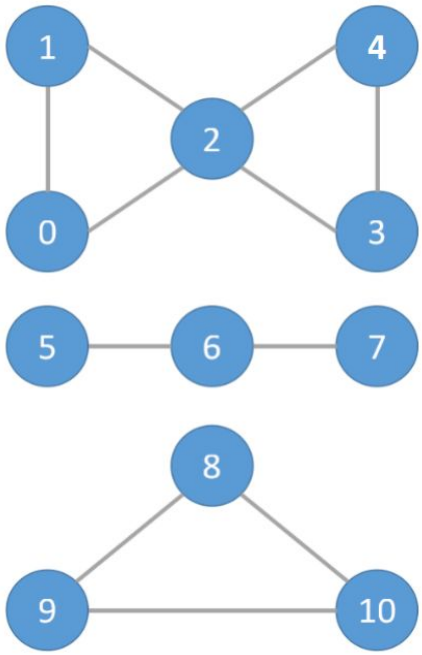
```
 seq | node_seq | node | scc
-----+----------+------+-----
   1 |        1 |    1 |   1
   2 |        2 |    2 |   1
   3 |        3 |    4 |   1
   4 |        1 |    0 |   2
```

```
   5 |       1 |    6 |   3
   6 |       1 |    5 |   4
   7 |       1 |    3 |   5
(7 rows)
```

Biconnected components

Example

Figure 3: example of biconnected components.



Demo slides

https://docs.google.com/presentation/d/1XjL5RvCXbLozcczPv6_IaEmjLx6xFGi1kWQQ_q-bXTo/present?slide=id.p

Use the BGL

Unlike connected components and strongly connected components, vertices may belong to multiple biconnected components.The better way is outputting edges instead of vertices.

Code

https://github.com/XJTUmg/ExampleCode/blob/master/BiconnectedComponents/BiconnectedComponents.cpp

Output

```
Total number of biconnected components 5
Edge 0 -- 1 is in biconnected component 1
Edge 1 -- 2 is in biconnected component 1
Edge 2 -- 0 is in biconnected component 1
Edge 2 -- 3 is in biconnected component 0
Edge 2 -- 4 is in biconnected component 0
```

```
Edge 3 -- 4 is in biconnected component 0
Edge 5 -- 6 is in biconnected component 3
Edge 6 -- 7 is in biconnected component 2
Edge 8 -- 9 is in biconnected component 4
Edge 9 -- 10 is in biconnected component 4
Edge 8 -- 10 is in biconnected component 4
```

Proposed function for pgRouting

Initial setup

```
CREATE TABLE edge_table (
      id BIGSERIAL,
      source BIGINT,
      target BIGINT,
      cost FLOAT,
      reverse_cost FLOAT
);

INSERT INTO edge_table (
      source, target,
      cost, reverse_cost) VALUES
(0, 1, 1, 1),
(1, 2, 1, 1),
(0, 2, 1, 1),
(2, 3, 1, 1),
(2, 4, 1, 1),
(3, 4, 1, 1),
(5, 6, 1, 1),
(6, 7, 1, 1),
(8, 9, 1, 1),
(9, 10, 1, 1),
(10, 8, 1, 1);
```

Procedure

I am proposing add `pgr_biconnectedComponents()` function to pgRouting.

```
SELECT * FROM pgr_biconnectedComponents (
      'SELECT id, source, target, cost, reverse_cost FROM edge_table');
```

Results

```
 seq | edge_seq | edge | edge_p1 | edge_p2 | bcc
-----+----------+------+---------+---------+-----
   1 |        1 |    4 |       2 |       3 |   1
   2 |        2 |    5 |       2 |       4 |   1
   3 |        3 |    6 |       3 |       4 |   1
   4 |        1 |    1 |       0 |       1 |   2
   5 |        2 |    2 |       1 |       2 |   2
```

```
   6 |        3 |    3 |       0 |       2 |   2
   7 |        1 |    7 |       5 |       6 |   3
   8 |        1 |    8 |       6 |       7 |   4
   9 |        1 |    9 |       8 |       9 |   5
  10 |        2 |   10 |       9 |      10 |   5
  11 |        3 |   11 |       8 |      10 |   5
(11 rows)
```

Analyze pgRouting Sample Data[9]

Use the BGL

Connected components: Code, Output.
Strongly connected components: Code, Output.
Biconnected components: Code, Output.

Initial setup

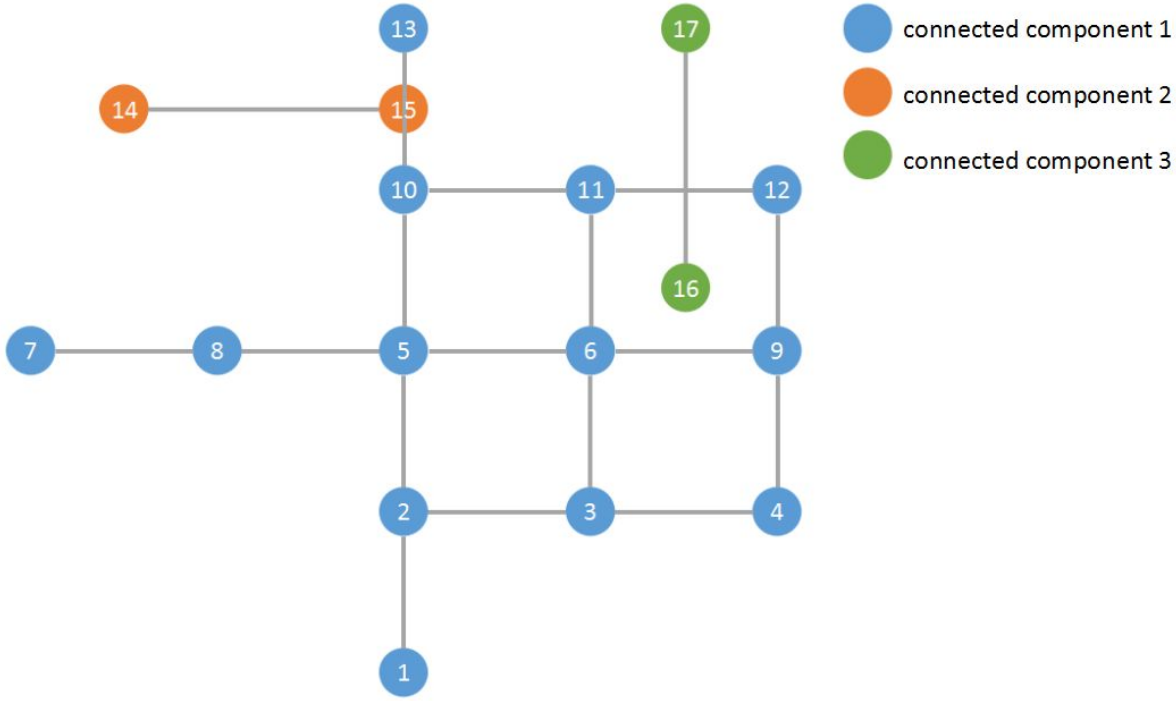
```
CREATE TABLE edge_table (
      id BIGSERIAL,
      source BIGINT,
      target BIGINT,
      cost FLOAT,
      reverse_cost FLOAT
);

INSERT INTO edge_table (
      source, target,
      cost, reverse_cost) VALUES
(1, 2, 1, 1),
(2, 3, -1, 1),
(3, 4, -1, 1),
(2, 5, 1, 1),
(3, 6, 1, -1),
(4, 9, 1, 1),
(5, 6, 1, 1),
(6, 9, 1, 1),
(7, 8, 1, 1),
(8, 5, 1, 1),
(5, 10, 1, 1),
(6, 11, 1, -1),
(9, 12, 1, 1),
(10, 11, 1, -1),
(11, 12, 1, -1),
(10, 13, 1, 1),
```

[9] pgRouting Sample Data.

```
(14, 15, 1, 1),
(16, 17, 1, 1);
```

Results

Connected components:
Figure 4: analyze pgRouting Sample Data using connected components algorithm.

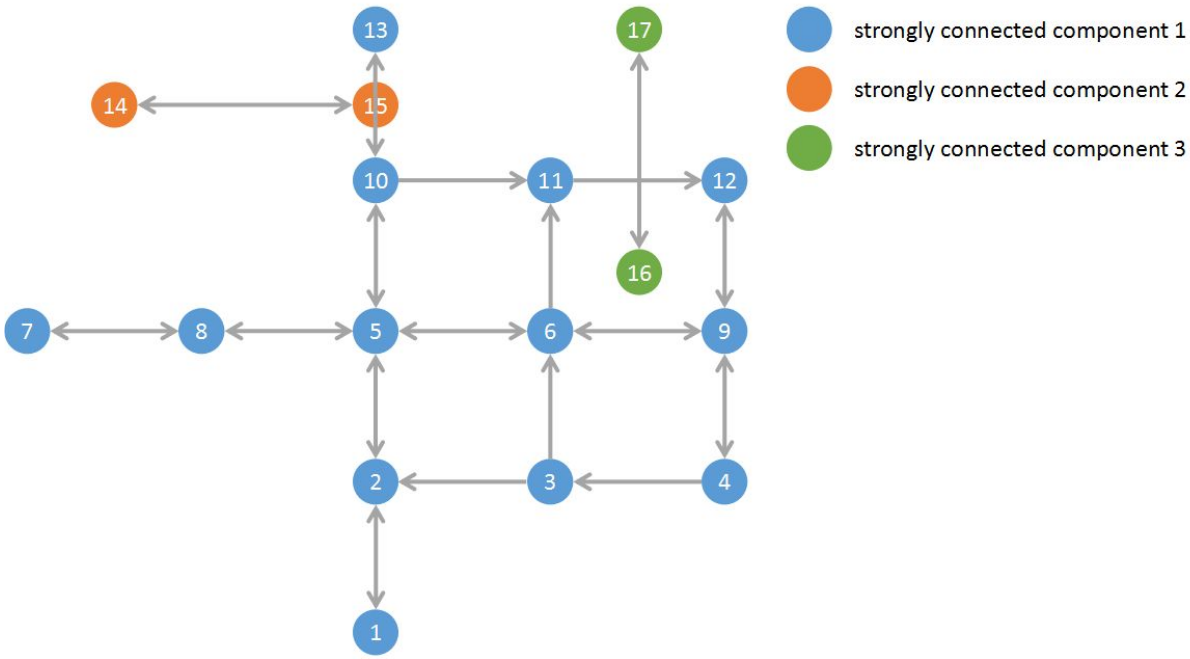


seq	node_seq	node	component
1	1	1	1
2	2	2	1
3	3	3	1
4	4	4	1
5	5	5	1
6	6	6	1
7	7	7	1
8	8	8	1
9	9	9	1
10	10	10	1
11	11	11	1
12	12	12	1
13	13	13	1
14	1	14	2
15	2	15	2
16	1	16	3
17	2	17	3

```
(17 rows)
```

Strongly Connected components:

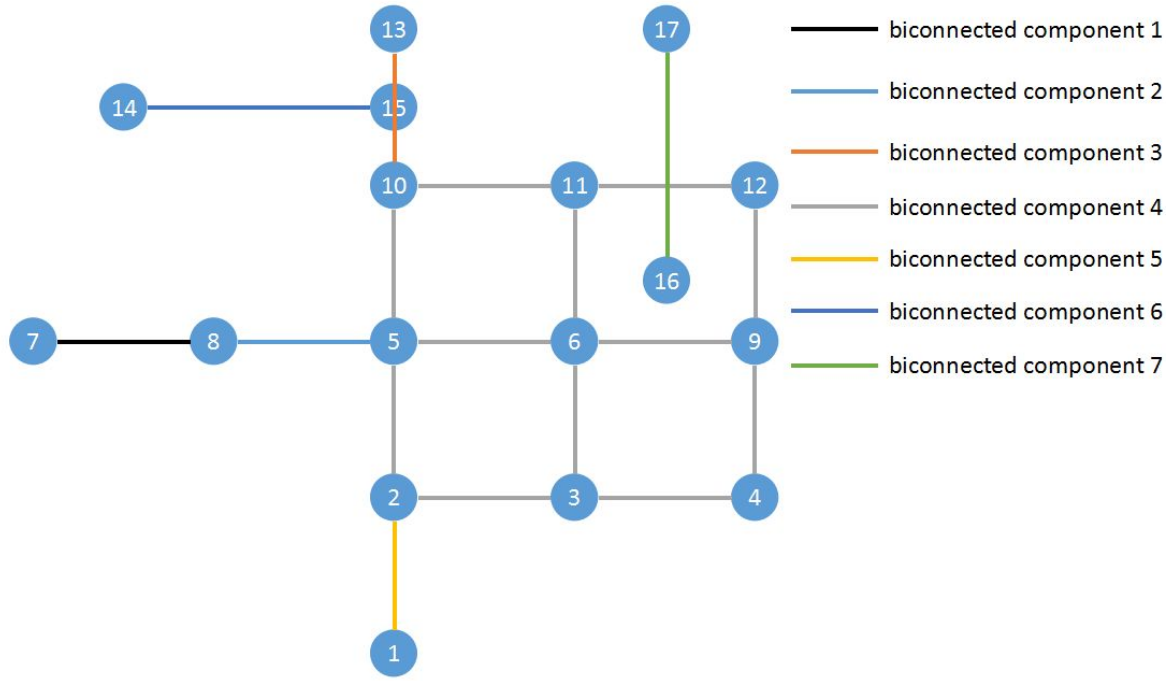
Figure 5: analyze pgRouting Sample Data using strongly connected components algorithm.




```
 seq | node_seq | node | scc
-----+----------+------+-----
   1 |        1 |    1 |   1
   2 |        2 |    2 |   1
   3 |        3 |    3 |   1
   4 |        4 |    4 |   1
   5 |        5 |    5 |   1
   6 |        6 |    6 |   1
   7 |        7 |    7 |   1
   8 |        8 |    8 |   1
   9 |        9 |    9 |   1
  10 |       10 |   10 |   1
  11 |       11 |   11 |   1
  12 |       12 |   12 |   1
  13 |       13 |   13 |   1
  14 |        1 |   14 |   2
  15 |        2 |   15 |   2
  16 |        1 |   16 |   3
  17 |        2 |   17 |   3
(17 rows)
```

Biconnected components:

Figure 6: analyze pgRouting Sample Data using biconnected components algorithm.




```
 seq | edge_seq | edge | edge_p1 | edge_p2 | bcc
-----+----------+------+---------+---------+-----
   1 |        1 |    9 |       7 |       8 |   1
   2 |        1 |   10 |       8 |       5 |   2
   3 |        1 |   16 |      10 |      13 |   3
   4 |        1 |    2 |       2 |       3 |   4
   5 |        2 |    3 |       3 |       4 |   4
   6 |        3 |    4 |       2 |       5 |   4
   7 |        4 |    5 |       3 |       6 |   4
   8 |        5 |    6 |       4 |       9 |   4
   9 |        6 |    7 |       5 |       6 |   4
  10 |        7 |    8 |       6 |       9 |   4
  11 |        8 |   11 |       5 |      10 |   4
  12 |        9 |   12 |       6 |      11 |   4
  13 |       10 |   13 |       9 |      12 |   4
  14 |       11 |   14 |      10 |      11 |   4
  15 |       12 |   15 |      11 |      12 |   4
  16 |        1 |    1 |       1 |       2 |   5
  17 |        1 |   17 |      14 |      15 |   6
  18 |        1 |   18 |      16 |      17 |   7
(18 rows)
```

13. Future Directions

Some functions which can be implemented for the future, see below:

1. Implement articulation points function for pgRouting by the BGL.
2. Implement disjoint-sets data structure and incremental connected components for pgRouting by the BGL.

3. Add a functionality to pgRouting for 2-SAT problem.
4. Add a functionality to pgRouting for finding bridges in an undirected graph.

For details:

1. **Articulation points**[10]: The `articulation_points()` functions in the BGL compute the articulation points of an undirected graph. Those vertices that belong to more than one biconnected components are called articulation points. The time complexity for the articulation points algorithm using Tarjan's algorithm based on DFS is also `O(V + E)`.
2. **Incremental connected components**[11]: Unlike `connected_components()`, in this situation, the graph is growing (edges are being added) and the connected components information needs to be updated repeatedly. The algorithm used here is based on the disjoint-sets[12] (fast union-find) data structure. The objects used here are a graph `g`, a disjoint-sets structure `ds`, and vertices `u` and `v`.
    a. The `initialize_incremental_components(g, ds)` function in the BGL does the basic initialization of the disjoint-sets structure.
    b. The `incremental_components(g, ds)` function in the BGL calculates the connected components in the graph *g* and the information is embedded in *ds*.
    c. The `ds.find_set(v)` function in the Boost extracts the component information for vertex *v* from the disjoint-sets.
    d. The `ds.union_set(u, v)` function in the Boost updates the disjoint-sets structure when edge `(u, v)` is added to the graph.

    The time complexity for the whole process is `O(V + E alpha(E, V))`.
3. **2-satisfiability problem**: We can solve 2-satisfiability instances in linear time[13] based on the notion of strongly connected components.
    a. Construct the graph of the instance, and find its strongly connected components.
    b. If any strongly connected components contains both a variable and its negation, then the instance is not satisfiable.
    c. Contract the strongly connected components, and then the condensation is automatically a directed acyclic graph.
    d. Topologically order the vertices of the condensation.
    e. Produce the results.
4. **Bridges in an undirected graph**: The algorithm[14] is similar to linear algorithm for articulation points. The time complexity for the algorithm is `O(V + E)`.

[10] the Boost Graph Library: Articulation Points - 1.46.0.
[11] the Boost Graph Library: Incremental Connected Components - 1.46.0.
[12] Galil Z, Italiano G F. Data structures and algorithms for disjoint set union problems[J]. ACM Computing Surveys (CSUR), 1991, 23(3): 319-344.
[13] Aspvall B, Plass M F, Tarjan R E. A linear-time algorithm for testing the truth of certain quantified boolean formulas[J]. Information Processing Letters, 1979, 8(3): 121-123.
[14] Finding bridges in a graph in O(N+M) - E-maxx Algorithms - MAXimal.