Depth First Search and Sequential Vertex Coloring for pgRouting

1. Contact Details

- **Name**: Ashish Kumar.
- **Country**: India.
- **Email**: ashishkr23438@gmail.com / ashishkumar.cse18@iitbhu.ac.in
- **Phone**: +91-6205144592
- **Location**: Varanasi, India, +5:30 GMT
- **Github**: https://github.com/krashish8
- **LinkedIn**: https://www.linkedin.com/in/ashishkr23438/

2. Title

Depth First Search and Sequential Vertex Coloring for pgRouting.

3. Brief Project Description

I propose to add the following algorithms in pgRouting during the GSoC ‘20 period:

Depth First Search algorithm is the implementation of a standard graph traversal algorithm for both *directed* graphs (those graphs in which every edge has a fixed direction) and *undirected* graphs (edges do not have any direction). It has been implemented in Boost Graph Library as two different algorithms - **Boost:: Depth First Search** for directed graphs and **Boost:: Undirected DFS** for undirected graphs. One of its applications is to find any path in the graph from a source vertex to all other vertices or to check if a vertex is connected to any other vertex in the graph. Also, it is used to find a topological sort of the graph, which can be used in scheduling a sequence of jobs or tasks based on their dependencies. It has a linear time complexity of `O(|V| + |E|)`.

Sequential Vertex Coloring is an algorithm to compute the vertex coloring of a graph. Vertex Coloring involves assigning colors to the vertices of a graph sequentially in such a way that no two adjacent vertices have the same color. It has been implemented in the Boost Graph Library as **Boost:: Sequential Vertex Coloring** for undirected graphs. One of its applications is to check if a graph is bipartite, i.e., to check whether a graph can be divided into two disjoint sets such that no edge connects vertices from the same set. It has a time complexity of `O(|V|*(|d| + |k|))`, where `|d|` and `|k|` are the maximum degree and the number of colors used in the graph, respectively.

Maximum Adjacency Search is an algorithm for undirected graphs **(is an additional idea for implementing during the GSoC period if time allows)** that performs a traversal of the vertices in the graph such that the next vertex visited will be that vertex which has the maximum visited neighbors at any time. It has been implemented in the Boost Graph Library as **Boost:: Maximum Adjacency Search** for undirected graphs. One of its applications is to compute a minimum cut of a graph, i.e., to find the minimum sum of weights of edges in a graph, which when removed from the graph divides the graph into two groups. It has a linear time complexity of `O(|V| + |E|)`.

4. State of the Project Before GSoC

pgRouting currently does not have these algorithms implemented.

Depth First Search is a standard graph searching algorithm and is used in various other already implemented algorithms such as Prim's and Kruskal's algorithm for finding MST. However, not a single standard function exists in pgRouting, either for directed graphs or for undirected graphs, i.e., pgRouting does not have it in the PostgreSQL functionality.

Sequential Vertex Coloring is not implemented before in pgRouting, and a single standard function does not exist.

Maximum Adjacency Search is also not implemented in pgRouting. The min-cut algorithm was implemented by Aditya Pratap Singh during GSoC 2018, however, this search algorithm wasn't implemented separately, which has several other applications too, apart from finding a minimum cut of the graph.

5. Benefits to the Community

These algorithms to be implemented are useful to users in various cases:

Implementing **Depth First Search** algorithm in pgRouting would enable the developers to use this algorithm wherever required with a single function call, instead of rewriting this algorithm every time. It would also allow the developers to perform a simple analysis of the graph since it is a searching algorithm which works on any kind of graph. Moreover, it can help in finding any path in a graph from a vertex to any other vertex or to check for the presence of cycles or bridges in a graph. In the case of directed graphs, it can be used to find the Strongly Connected Components (A strongly connected component is a subgraph such that one can reach from any vertex to any other vertex in the subgraph) or to find a topological sort of the graph. Finding a strongly connected component in a geographic map or a topological sorting of dependent jobs can be of great use to people in daily life. It can also be used to compute the reachability and to detect cycles in a graph.

The **Sequential Vertex Coloring** algorithm has several benefits to the community. As an example, it provides a solution to any scheduling problem, be it job scheduling or exam scheduling problem. Scheduling the exams such that there is no clash when two exams which a student has opted for are scheduled at the same time, can be solved by coloring the graph, with vertex being the exams and edges being a common student with two exams. Coloring a geographical map is also an application of Sequential Vertex Coloring, in which we shall not assign two adjacent maps with the same color.

Implementing the **Maximum Adjacency Search** algorithm in pgRouting would simplify the implementation of the Minimum Cut algorithm already implemented because this search algorithm can then be used by a single function call, which will make the code of Minimum Cut easier to read and understand. It has other applications too such as finding a perfect elimination ordering or finding minimal triangulations in a graph, which can also be applied to many real-life graphical networks.

6. Deliverables

1. Implementation of Depth First Search algorithm for pgRouting: For that, the function `pgr_depthFirstSearch()` must be constructed which will be applicable for both directed and undirected graphs, and it will return a possible ordering of the vertices of the graph during depth-first search traversal.

2. Implementation of the Sequential Vertex Coloring algorithm for pgRouting: For that, the function `pgr_sequentialVertexColoring()` must be constructed, and it will return the colors to be assigned to the vertices of the graph, in sequential order.

Each implemented function will be delivered with the relevant documentation and the tests included.

7. Timeline

Community Bonding Period
(May 4th - June 1st)

- ➢ Set up the development environment.
- ➢ Interact with mentors, introduce myself to the community, and actively get involved in the discussion.
- ➢ Get familiar with pgRouting's development style. Understand expected coding, documentation, and testing standards set by pgRouting.
- ➢ Set up the wiki page to keep track of weekly progress.
- ➢ Develop a better understanding of PostgreSQL, PostGIS, Pl/pgSQL, and how they interact with pgRouting.
- ➢ Learn to create unit tests using pgTap.
- ➢ Implement simple dummy functions to understand pgRouting better.

First Coding Period
(June 1st - June 29th)

Time Period	Proposed Work
Week 1 (June 1st - June 8th)	➢ Developing `pgr_depthFirstSearch()` starts. ➢ Create a basic skeleton for C, C++, SQL code and for documentation and tests.
Week 2 (June 8th - June 15th)	➢ Read data from PostgreSQL. ➢ Transform data to C++ containers suitable for using with Boost.
Week 3 (June 15th - June 22nd)	➢ Create the necessary class wrappers for the Boost function. ➢ Process the data with the Boost function. ➢ Transform results to C containers suitable for passing to PostgreSQL.
Week 4	➢ Prepare user documentation.

(June 22nd - June 29th)	➢ Create suitable queries using the sample data of the pgRouting documentation. ➢ Create the first term report.

➢ First Evaluation Period (June 29th - July 3rd):

- ○ Submit working `pgr_depthFirstSearch()` function (albeit without pgTap tests).
- ○ Mentors evaluate me and vice-versa.

Second Coding Period
(June 29th - July 27th)

Time Period	Proposed Work
Week 1 **(June 29th - July 6th)**	➢ Developing `pgr_sequentialVertexColoring()` starts. ➢ Create a basic skeleton for C, C++, SQL code and for documentation and tests.
Week 2 **(July 6th - July 13th)**	➢ Read data from PostgreSQL. ➢ Transform data to C++ containers suitable for using with Boost.
Week 3 **(July 13th - July 20th)**	➢ Create the necessary class wrappers for the Boost function. ➢ Process the data with the Boost function. ➢ Transform results to C containers suitable for passing to PostgreSQL.
Week 4 **(July 20th - July 27th)**	➢ Prepare user documentation. ➢ Create suitable queries using the sample data of the pgRouting documentation. ➢ Create the first term report.

➢ Second Evaluation Period (July 27th - July 31st):

- ○ Submit working `pgr_sequentialVertexColoring()` function (albeit without pgTap tests).
- ○ Mentors evaluate me and vice-versa.

Third Coding Period
(July 27th - August 24th)

Time Period	Proposed Work
Week 1 **(July 27th - August 3rd)**	➢ Tests for function `pgr_depthFirstSearch().` ○ create pgTap tests to check no server crash. ○ create pgTap unit tests for expected results for different small graphs: ■ one vertex graph ■ one edge graph ■ two edge graph

	■ cycle graph with 3 edges ➢ Work on the feedback provided from the second evaluation. ➢ Basic implementation of the function. ➢ Basic testing.
Week 2 (August 3rd - August 10th)	➢ Tests for function `pgr_sequentialVertexColoring().` ○ create pgTap tests to check no server crash. ○ create pgTap unit tests for expected results for different small graphs: ■ one vertex graph ■ one edge graph ■ two edge graph ■ cycle graph with 3 edges ➢ Work on the feedback provided from the second evaluation. ➢ Basic implementation of the function. ➢ Basic testing.
Week 3 (August 10th - August 17th)	➢ Integration to the `develop branch` in the main repository.
Week 4 (August 17th - August 24th)	➢ Preparation of final report

- **Final Evaluation Period (August 24th - August 31st):**
  - Submit a detailed final report with all the required functions, documentation, and unit tests.
  - Submit final report and evaluation of mentors.

8. Do you understand this is a serious commitment, equivalent to a full-time paid summer internship or summer job?

Yes, I completely understand that GSoC is a serious commitment, and the expectations from me would be very similar to a full-time paid summer internship or summer job. Therefore, I will try to put my best efforts to make worthy contributions to the pgRouting community during GSoC, and even after that, if time permits. Also, similar to a job, I'd be expected to respond frequently to the daily updates and reports on a weekly basis. I am really excited to work on this project.

9. Do you have any known time conflicts during the official coding period?

No, I do not have any large time conflicts during the official coding period.

However, my college will reopen sometime in the middle of the Third Coding Period, so I'll have to devote around 4-5 hours daily to the classes, and the rest of the time can be given to my GSoC work. I can assure that it won't affect my performance much, because the initial weeks of a semester in my college are generally not very hectic.

10. Studies

10.1 What is your school and degree?

School: Indian Institute of Technology, Banaras Hindu University (BHU), Varanasi
(IIT (BHU) Varanasi).
Degree: Bachelor of Technology in Computer Science and Engineering.

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

Yes, my application will contribute to my ongoing studies and Bachelor's Degree.

I enjoy coding and exploring various algorithms and have previously participated in various algorithmic competitions, so this project will help me get a practical experience of algorithmic coding. It would be an added asset to my ongoing degree and would help me gain a deeper understanding of GIS. I will get hands-on experience of the algorithms implemented in pgRouting, and the experience and information obtained would help me contribute a lot to my B.Tech. Thesis / Project that I'll take in my forthcoming semesters.

11. Programming and GIS

11.1 Computing experience:

- **Operating System:** Ubuntu 19.10 (Eoan) - Used on a Daily Basis, Windows 10.
- **Programming Languages:** C++, C, Python3, Javascript, Java, PHP, Bash.
- **Databases**: PostgreSQL, MySQL.
- **Tools:** Git, Jupyter Notebook.
- **Frameworks:** Django, Django Rest Framework, Angular, Vue.js
- **Relevant Courses Completed:** Data Structures and Algorithms, Graph Theory.

11.2 GIS experience as a user:

I have used GIS libraries like pgRouting and have used GIS before on a database management project.

11.3 GIS programming and other software programming:

- ❖ Worked on a database management application written in Django, SQL, HTML/JS/CSS.
- ❖ Worked on a Slack Bot app and a backend web app written in Django Rest Framework for Technex '20 hackathon. Also, this app won me the 3rd position in the hackathon.
- ❖ Worked on a web app written in SQL, PHP, HTML/JS/CSS, which was extensively used in our college for the smooth organization of our college's Sports Fest.

- ❖ Worked on a Django REST API project written in Django REST Framework for making the Backend of our college's technical fest, which was used by ~8000 people.
- ❖ Public C / C++ related programming / projects:
  - ➢ I have made around 1000+ code submissions on various online competitive programming platforms such as Codeforces, Codechef, HackerRank, Spoj. Submissions also include the graph problems and the usage of the graph algorithms in solving those problems.
  - ➢ I maintain GitHub repositories containing my submitted C++ Online Judge solutions and C++ algorithm templates used by me frequently.
  - ➢ I maintain repositories containing my codes submitted during the Data Structures and Algorithms course, Algorithms course and Operating System course at IIT (BHU) Varanasi.
- ❖ I'm an enthusiastic Open Source contributor (GitHub Profile).

12. GSoC Participation

12.1 Have you participated in GSoC before?

No, I have not participated in GSoC before. This year will be my first time applying.

12.2 Have you submitted / will you submit another proposal for this year's GSoC to a different org?

No.

13. pgRouting Application Requirements

The requirements for applying to pgRouting (under OSGeo) are mentioned here:
https://github.com/pgRouting/pgrouting/wiki/GSoC-Ideas%3A-2020#pgrouting-application-requirements
The links to the respective issues are:

- Issue: Have experience with GitHub & Git → https://github.com/krashish8/GSoC-pgRouting/issues/1
- Issue: Build locally pgRouting → https://github.com/krashish8/GSoC-pgRouting/issues/2
- Issue: Get familiar with C++ → https://github.com/krashish8/GSoC-pgRouting/issues/3

14. Detailed Proposal

14.1 Depth First Search[2][3] (From Boost Graph Library[1])

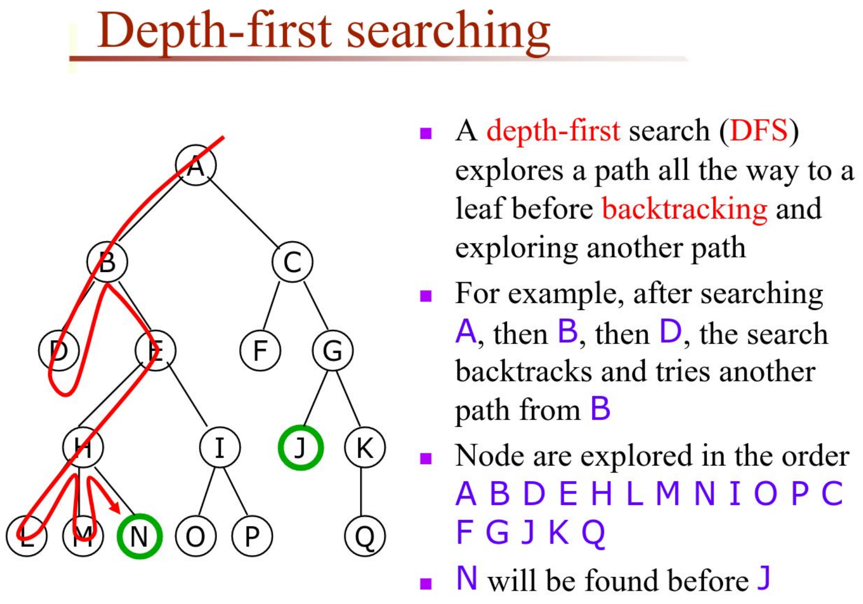
Depth First Search[4] is a standard graph traversal algorithm that can be applied both on directed as well as undirected graphs, and the graph can be either weighted or unweighted. An undirected graph is a graph in which the edges do not have any fixed direction, whereas a directed graph is the one in which every edge has a fixed direction. Also, a weighted graph is the one in which every edge has a weight or cost associated with it, whereas the edges in an unweighted graph have no weights, i.e., all the edges have the same cost.

The algorithm depth-first search starts at any arbitrary vertex of the graph, known as the *source* vertex. It then visits all other vertices of the graph in a sequential order that are *reachable* from the source vertex selected in the first step. A vertex V is said to be reachable from vertex S if and only if there exists a path or a sequence of adjacent vertices that starts with S and ends with V.

The depth-first search traversal proceeds in the following manner: the vertices are visited in a strictly increasing order of their distance from the source vertex until a vertex is reached with no adjacent vertices or with all visited adjacent vertices. Here, the distance of a vertex denotes the number of edges in the path from the source vertex when the cost of an edge is ignored. In case of 'directed' graphs, in order to move from a vertex *u* to another vertex *v*, the edge connecting them must point from *u* to *v*. If the edge points in the opposite direction, then one cannot reach directly from u to v.

Then the algorithm backtracks until there is a vertex with some unvisited adjacent vertices. If such a vertex exists, then that vertex is visited next. The algorithm proceeds recursively until all the reachable vertices are visited. Then another source vertex is selected from the rest of the unvisited vertices, and traversal is continued from that vertex. The traversal terminates when all the vertices of the graph have been visited. DFS Traversal of a tree-like graph is shown in Fig 14.1.1.

This algorithm has been already implemented in the Boost Graph Library. The documentation for the undirected version can be found here, and the source code can be found here, and the documentation of the directed version of depth first search can be found here, and the source code can be found here.



Depth-first searching

- A depth-first search (DFS) explores a path all the way to a leaf before backtracking and exploring another path
- For example, after searching A, then B, then D, the search backtracks and tries another path from B
- Node are explored in the order A B D E H L M N I O P C F G J K Q
- N will be found before J

Fig 14.1.1: Depth First Search Traversal of a tree-like graph [Source]

14.1.1 Algorithm

```
// Depth First Search from a source vertex s
procedure Depth-First-Search(G, s)[2][3]:
        color[s] := GRAY                                // Processing vertex
        entry_time[s] := time
        time := time + 1
```

```
    for each vertex t in Adj[s]:
        if color[t] = WHITE:                // Unvisited Vertex (Tree Edge)
                parent[t] := s
                call Depth-First-Search(G, t)
        else if color[t] = GRAY:
                cycle_exist := TRUE         // Back Edge
        else if color[t] = BLACK and time[t] > time[s]:
                // Forward Edge
        else if color[t] = BLACK and time[t] <= time[s]:
                // Cross Edge
    color[s] := BLACK                       // Processed vertex
    exit_time[s] := time
    time := time + 1
```

This algorithm has a linear time complexity of O(|V| + |E|), where |V| and |E| denote the total number of vertices and edges in the graph, respectively.

In short, the idea of Depth First Search is to go as deep as possible from the source vertex and backtrack once a vertex is reached with no adjacent vertices or with all visited adjacent vertices. An important observation is that two graphs that differ only in the weights of their edges would produce the same result with the depth-first search algorithm. Hence, a depth-first search gives a simple analysis of the graph, i.e., whether two graphs have the same number and ordering of vertices, irrespective of the costs of their edges. Undirected DFS traversal is shown in Fig 14.1.2 and Fig 14.1.3, and directed DFS traversal is shown in Fig 14.1.4 and Fig 14.1.5.

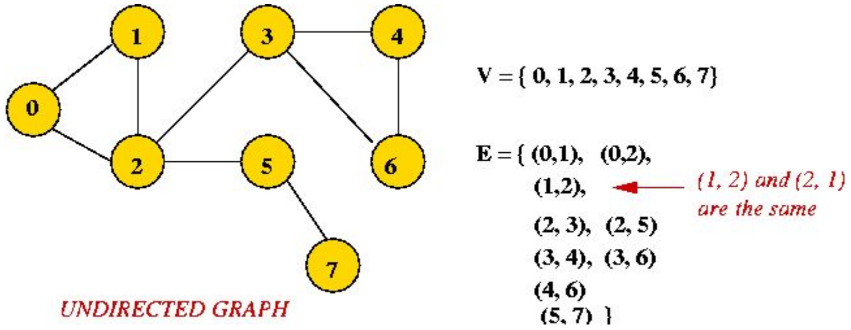


Fig 14.1.2: Representation of an undirected graph [Source]

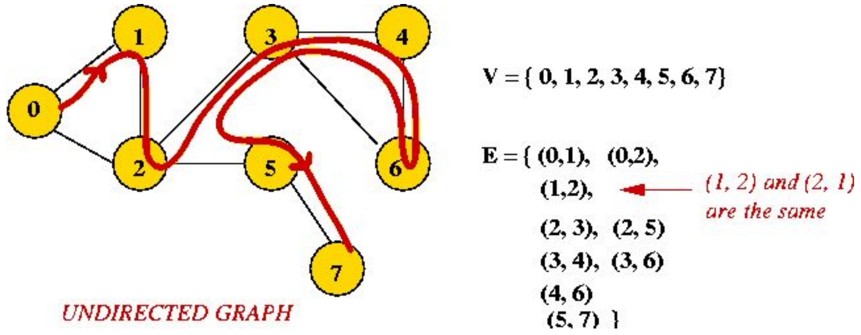


Fig 14.1.3: Undirected depth-first search traversal of the given undirected graph

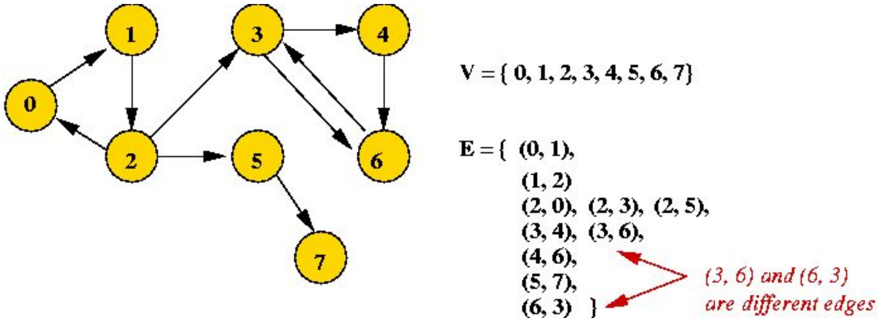


Fig 14.1.4: Representation of a directed graph [Source]

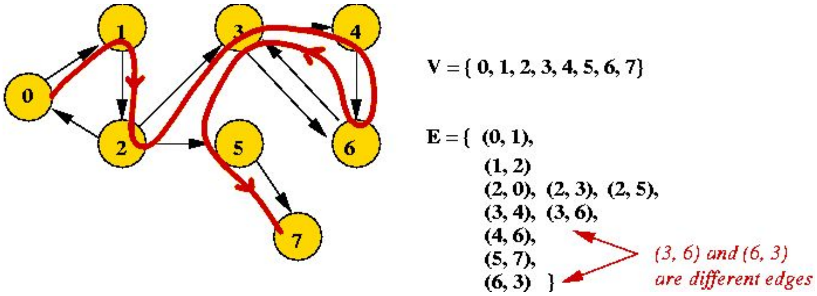


Fig 14.1.5: Depth-first search traversal of the given directed graph

14.1.2 Applications

The Depth First Search algorithm in a graph has a variety of applications. It can be used to:

- Find any path from the source vertex to all other vertices (path may not exist, and it may not be of minimal length). Therefore, it can help a person to check whether it is possible to go from one place to another.
- Find a topological sorting of a *directed* graph. For this, a series of depth-first searches are made on the graph, and the required topological sorting will be the vertices arranged in decreasing order of their exit time.
- Find strongly connected components in a *directed* graph. To find strongly connected components, first, a topological sorting of the graph is found, and then a series of depth-first searches are made on the transposed graph in the order defined by the topological sort. Each depth-first search creates a strongly connected component.
- Find the ancestor of a vertex or to find the least common ancestors of two or more vertices in a tree-like directed graph.
- Check whether a graph is *acyclic* or not, and find the cycles in a graph. A cyclic graph is a graph in which some number of vertices are connected in a closed chain. A graph that isn't cyclic is acyclic.

  - To check this, run the DFS traversal of the graph. For every visited vertex *v*, if there exists an adjacent vertex *u* which has been already visited but is not a parent of *v*, then there exists a cycle in the graph.
  - If no such vertex is found, then the graph is acyclic.
    (Assumption: there are no parallel edges in the graph)

- Find *bridges* in an *undirected* graph. A *bridge* is an edge of the graph whose removal increases the number of connected components. To find a bridge, run the DFS traversal of the graph and convert it into a directed graph in the same order as the traversal moves through the vertices. Then, find Strongly Connected Components in the graph. A bridge is that edge whose endpoints belong to different connected components. Bridges are weak edges in the graph, and therefore, they can help one deduce the weak links in any graphical network.

14.1.3 Proposed File Structure

NOTE: The proposed file structure may change once the implementation begins.

Directory	Files
`doc/depthFirstSearch/`	`CMakeLists.txt` `pgr_depthFirstSearch.rst`

`docqueries/depthFirstSearch/`	`CMakeLists.txt`
	`doc-pgr_depthFirstSearch.result`
	`doc-pgr_depthFirstSearch.test.sql`
	`test.conf`

`include/depthFirstSearch/`	`pgr_depthFirstSearch.hpp`
`include/drivers/depthFirstSearch/`	`depthFirstSearch_driver.h`

`pgtap/depthFirstSearch/depthFirstSearch/`	`depthFirstSearch-edge-cases.sql`
	`depthFirstSearch-innerQuery.sql`
	`depthFirstSearch-types-check.sql`
	`no_crash_test-depthFirstSearch.sql`

`sql/depthFirstSearch/`	`CMakeLists.txt`
	`_depthFirstSearch.sql`
	`depthFirstSearch.sql`

`src/depthFirstSearch/`	`CMakeLists.txt`
	`depthFirstSearch.c`
	`depthFirstSearch_driver.cpp`

14.1.4 Proposed Signature

NOTE: The proposed signatures and parameters may change once the implementation of this function begins.

The possible variants are:

- `pgr_depthFirstSearch()` Single vertex

```
pgr_depthFirstSearch(edges_sql, root_vid [, max_depth] [, directed])
RETURNS SET OF (seq, depth, start_vid, node, edge, cost, agg_cost)
OR EMPTY SET
```

- `pgr_depthFirstSearch()` Multiple vertices

```
pgr_depthFirstSearch(edges_sql, root_vids [, max_depth] [, directed])
RETURNS SET OF (seq, depth, start_vid, node, edge, cost, agg_cost)
OR EMPTY SET
```

`EMPTY SET` is returned when no path exists or the root vertex is not present in the graph.

Parameters[10]

Parameter	Type	Description
edges_sql	`TEXT`	Inner SQL query, as described below.
root_vid	`BIGINT`	Identifier of the root vertex of the tree. • Used on Single vertex.
root_vids	`ARRAY[ANY-INTEGER]`	Array of identifiers of the root vertices. • Used on Multiple vertices. • For optimization purposes, any duplicated value is ignored.

Optional Parameters

Parameter	Type	Default	Description
max_depth	`BIGINT`	9223372036854775807	Upper limit for depth of node in the tree - When value is `Negative` then **throws error**
directed	`BOOLEAN`	`true`	- When `true` Graph is considered Directed - When `false` the graph is considered as Undirected.

Inner Query[10]

edges_sql: It should be 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.

Here,

`ANY-INTEGER` = SMALLINT, INTEGER, BIGINT
`ANY-NUMERICAL` = SMALLINT, INTEGER, BIGINT, REAL, FLOAT

Result Columns

Returns set of `(seq, depth, start_vid, node, edge, cost, agg_cost)`

Column	Type	Description
seq	`BIGINT`	Sequential value starting from **1**.
depth	`BIGINT`	Depth of the `node`. ● 0 when `node` = `start_vid`.
start_vid	`BIGINT`	Identifier of the root vertex. ● In Multiple vertices, results are in ascending order.
node	`BIGINT`	Identifier of `node` reached using `edge`.
edge	`BIGINT`	Identifier of the edge used to arrive to `node`. ● −1 when `node` = `start_vid`.
cost	`FLOAT`	Cost to traverse `edge`.
agg_cost	`FLOAT`	Aggregate cost from `start_vid` to `node`.

14.1.5 Usage

Graph data is taken from [pgRouting Sample Data](#).

```
SELECT id, source, target, cost, reverse_cost FROM edge_table;

 id | source | target | cost | reverse_cost
----+--------+--------+------+--------------
  1 |      1 |      2 |    1 |            1
  2 |      2 |      3 |   -1 |            1
  3 |      3 |      4 |   -1 |            1
  4 |      2 |      5 |    1 |            1
  5 |      3 |      6 |    1 |           -1
  6 |      7 |      8 |    1 |            1
  7 |      8 |      5 |    1 |            1
  8 |      5 |      6 |    1 |            1
  9 |      6 |      9 |    1 |            1
 10 |      5 |     10 |    1 |            1
 11 |      6 |     11 |    1 |           -1
 12 |     10 |     11 |    1 |           -1
 13 |      11 |      12 |     1 |            -1
 14 |      10 |      13 |     1 |             1
 15 |       9 |      12 |     1 |             1
 16 |       4 |       9 |     1 |             1
 17 |      14 |      15 |     1 |             1
 18 |      16 |      17 |     1 |             1
(18 rows)
```

Query I (For Directed Graphs)

Find the Depth First Search Traversal of the *directed* graph with root vertex 1.

```
SELECT * FROM pgr_depthFirstSearch(
    ‘SELECT id, source, target, cost, reverse_cost FROM edge_table
     ORDER BY id’,
     1
);
```

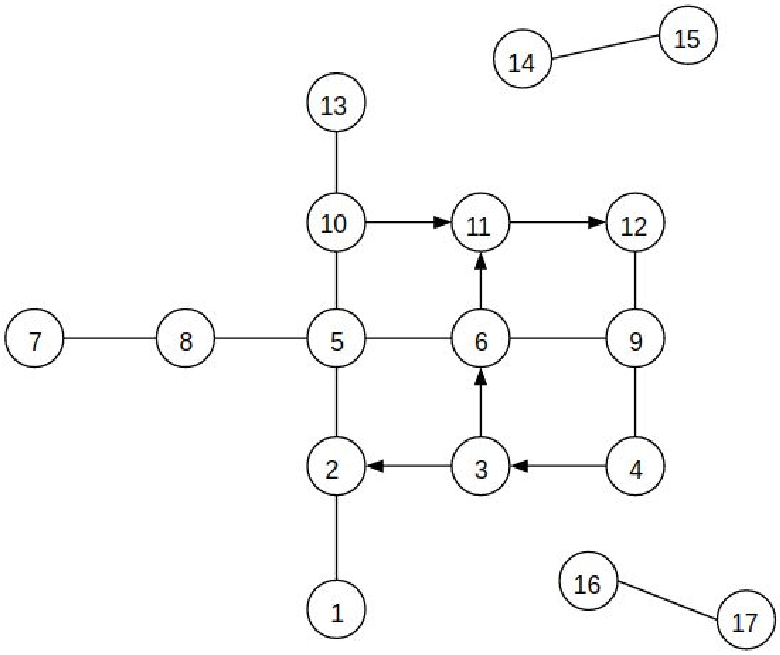


Fig 14.1.6: Directed graph representation of pgRouting Sample Data[7]
In this representation, bidirectional edges are represented by a *line*
whereas directional edges are represented by an *arrow*.

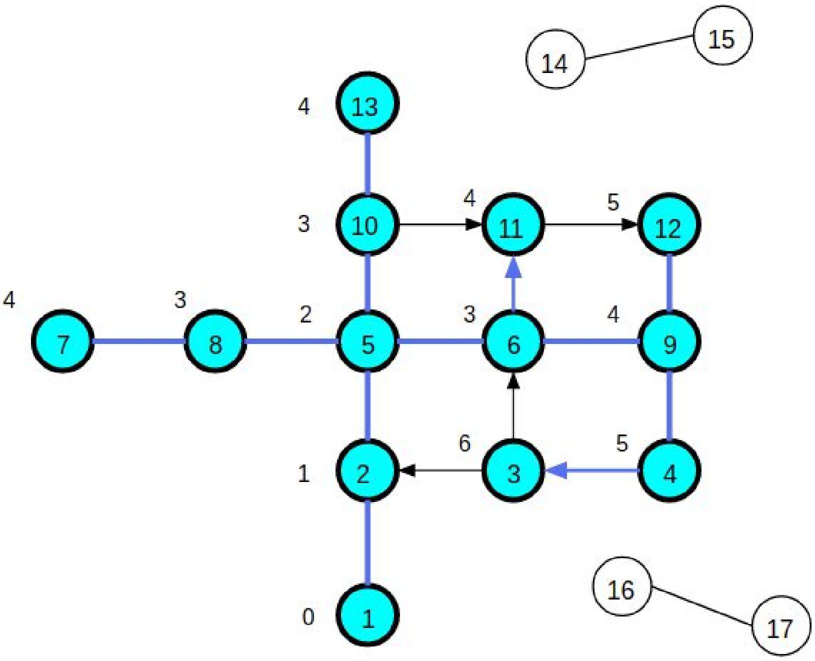


Fig 14.1.7: Final State of the directed graph after Depth First Search Traversal, taking root vertex as 1. Numbers outside the vertex denote the aggregate cost of visiting the vertex through DFS traversal.

Note: If multiple vertices are adjacent to a given vertex, then in this implementation, that vertex is visited first, which comes first in the `edge_table`.

Order of Traversal of vertices:

```
1 → 2 → 5 → 8 → 7 → 6 → 9 → 12 → 4 → 3 → 11 → 10 → 13
```

Output I (For Directed Graphs)

```
 seq | depth | start_vid | node | edge | cost | agg_cost
-----+-------+-----------+------+------+------+----------
   1 |     0 |         1 |    1 |   -1 |    0 |        0
   2 |     1 |         1 |    2 |    1 |    1 |        1
   3 |     2 |         1 |    3 |    2 |    1 |        2
   4 |     3 |         1 |    4 |    3 |    1 |        3
   5 |     4 |         1 |    9 |   16 |    1 |        4
   6 |     5 |         1 |    6 |    9 |    1 |        5
   7 |     6 |         1 |    5 |    8 |    1 |        6
   8 |     7 |         1 |    8 |    7 |    1 |        7
   9 |     8 |         1 |    7 |    6 |    1 |        8
  10 |     7 |         1 |   10 |   10 |    1 |        7
  11 |     8 |         1 |   11 |   12 |    1 |        8
  12 |     9 |         1 |   12 |   13 |    1 |        9
  13 |     8 |         1 |   13 |   14 |    1 |        8
(13 rows)
```

Query II (For Undirected Graphs)

Find the Depth First Search Traversal of the *undirected* graph with root vertex 1.

```
SELECT * FROM pgr_depthFirstSearch(
    ‘SELECT id, source, target, cost, reverse_cost FROM edge_table
     ORDER BY id’,
     1, directed := false
);
```

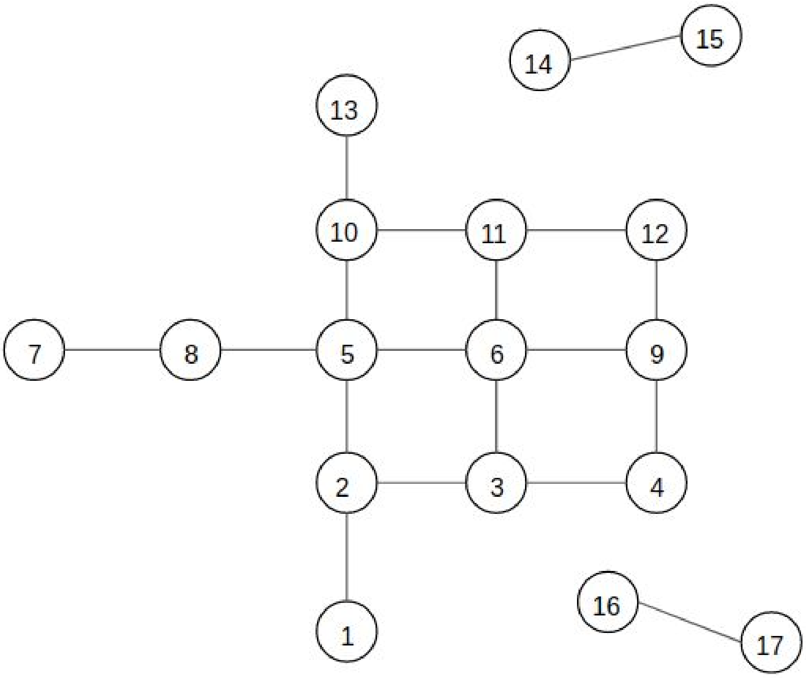


Fig 14.1.8: Undirected graph representation of pgRouting Sample Data[7]

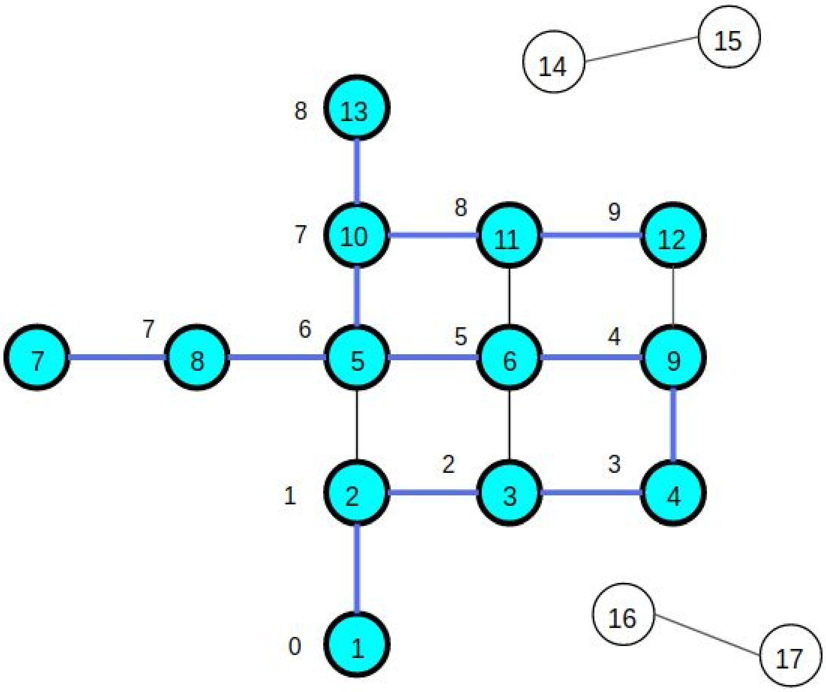


Fig 14.1.9: Final State of the undirected graph after Depth First Traversal, taking root vertex as 1. Numbers outside the vertex denote the aggregate cost of visiting the vertex through DFS traversal.

Note: If multiple vertices are adjacent to a given vertex, then in this implementation, that vertex is visited first, which comes first in the `edge_table`.

Order of Traversal of vertices:

`1 → 2 → 3 → 4 → 9 → 6 → 5 → 8 → 7 → 10 → 11 → 12 → 13`

Output II (For Undirected Graphs)

```
 seq | depth | start_vid | node | edge | cost | agg_cost
-----+-------+-----------+------+------+------+----------
   1 |     0 |         1 |    1 |   -1 |    0 |        0
   2 |     1 |         1 |    2 |    1 |    1 |        1
   3 |     2 |         1 |    5 |    4 |    1 |        2
   4 |     3 |         1 |    8 |    7 |    1 |        3
   5 |     4 |         1 |    7 |    6 |    1 |        4
   6 |     3 |         1 |    6 |    8 |    1 |        3
   7 |     4 |         1 |    9 |    9 |    1 |        4
   8 |     5 |         1 |   12 |   15 |    1 |        5
   9 |     5 |         1 |    4 |   16 |    1 |        5
  10 |     6 |         1 |    3 |    3 |    1 |        6
  11 |     4 |         1 |   11 |   11 |    1 |        4
  12 |     3 |         1 |   10 |   10 |    1 |        3
  13 |     4 |         1 |   13 |   14 |    1 |        4
(13 rows)
```

Note: Aggregate cost in a depth-first search traversal from source vertex to destination vertex is not always minimal.

14.1.6 Visualization

Animation of the Depth First Search Traversal (Directed)
https://github.com/krashish8/GSoC-pgRouting/issues/7#issuecomment-597593503
(External link because GIFs cannot be embedded in PDF format)

Animation of the Undirected Depth First Search Traversal
https://github.com/krashish8/GSoC-pgRouting/issues/7#issuecomment-597590597
(External link because GIFs cannot be embedded in PDF format)

Node / Edge Representation	Meaning
○	Unvisited vertex
⭕	Currently processing vertex
◯	Visited vertex
🔵	Visited and processed vertex
─	Unvisited edge
━	Visited edge
▬	Processed edge (while backtracking)

14.2 Sequential Vertex Coloring[5] (From Boost Graph Library[1])

Sequential Vertex Coloring or **Graph Coloring**[6] is a graph labeling algorithm in which labels or colors are assigned to the vertices of a graph such that no edge connects two identically labeled or colored vertices. Since the vertices are colored in a sequential order, this algorithm is also called the Sequential Vertex Coloring algorithm, and it is applicable for undirected graphs.

Mathematically, vertex coloring of a graph with vertices $v_1, v_2, \ldots, v_n$ assigns the smallest possible color to v_i, where the color of a vertex can be denoted by a non-negative integer.
Consider a set S denoting the colors of a vertex, i.e., $S = \{0, 1, 2, 3, \ldots, |V|-1\}$.
Then for a graph $G = (V, E)$, sequential vertex coloring is a map $c: V \rightarrow E$, such that $c(u) \neq c(v)$, for all edge (u, v) in G. It is evident that a maximum of $|V|$ colors (ranging from 0 to $|V| - 1$) are required to color all the vertices of the graph with all the conditions fulfilled, where |V| denotes total number of vertices in the graph. Fig 14.2.1 shows a proper coloring of the graph, using Sequential Vertex Coloring algorithm. Here, proper coloring means that no two vertices connected by an edge have the same color.

There are several other Graph coloring algorithms like *Edge Coloring*, in which edges of a graph are colored such that no two edges incident at a vertex have the same color or *Face Coloring*, in which faces or regions of a planar graph are colored such that no two adjacent faces have the same color. However, an important observation is that all these algorithms are reducible to the standard vertex coloring algorithm, as follows:

- Edge coloring of a graph is equivalent to the vertex coloring of its equivalent *line graph* (edges of the graph are transformed to vertices, and those vertices of line graph which share a common endpoint are connected by an edge).
- Face coloring of a *(planar)* graph is equivalent to the vertex coloring of its *dual* (faces are transformed to vertices, and adjacent faces are connected by an edge).

Thus, it is sufficient to study and implement Vertex Coloring algorithm, because all other similar Graph Coloring algorithms could be easily derived from it.

Color	Value
Yellow	0
Blue	1
Green	2

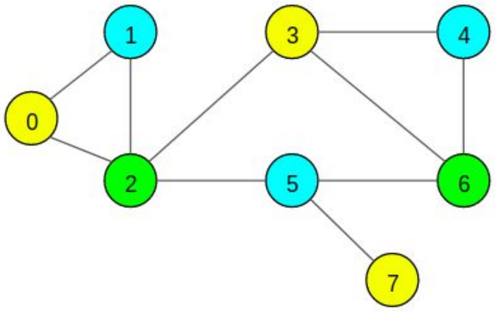


Fig 14.2.1: Minimum three colors are required for a proper coloring of the graph.

The Sequential Vertex Coloring algorithm proceeds in the following manner: first, all the vertices of the graph are initialized with the maximum possible color $|V| - 1$. Then, all the vertices of the graph are iterated sequentially, and for every vertex, the smallest possible color that is not used by its neighbors is assigned to it. Hence, proceeding in this manner, every vertex will be assigned the minimum possible color along with fulfilling the condition that every two vertices joined by an edge have different colors. Thus, a greedy strategy is followed in this algorithm.

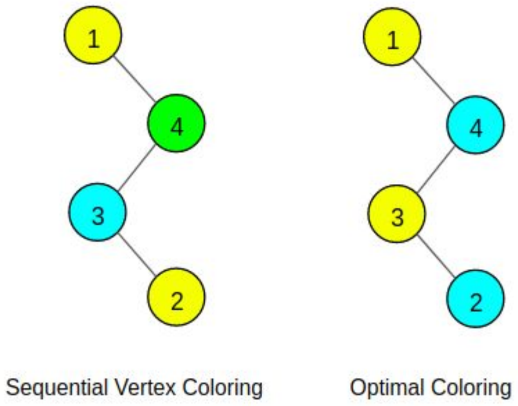
This algorithm has been already implemented in the Boost Graph Library. The documentation can be found here, and the source code can be found here.

14.2.1 Algorithm

```
procedure Sequential-Vertex-Coloring(G)[5]:
    max_color := 0
    Declare mark[|V|]                // contains mark of each color
    for each vertex u in V(G):
         color[u] := |V| - 1         // initialize color of vertex
    for i in range 0 … (|V| - 1):
         current_vertex := V[i]
         for each vertex v in Adj[current_vertex]:
                mark[color[v]] := i // mark the color of adjacent vertices
                                    // value of marking := i
         j := 0
         while j < max_color and mark[j] = i:
                j := j + 1
         if j = max_color:
                max_color := max_color + 1
         color[current_vertex] := j
```

This algorithm has a linear time complexity of O(|V|*(|d| + |k|)), where |d| and |k| denote the maximum degree and the number of colors used in the graph, respectively.

Though this algorithm provides a method for a proper coloring of the graph, yet it is not optimal, as shown in Fig 14.2.2. The *Chromatic number* of a graph is the minimum number of colors required for a proper coloring of the graph. There is no efficient algorithm to calculate the chromatic number, and thus it is an NP-Hard Problem.



Sequential Vertex Coloring

Optimal Coloring

Fig 14.2.2: Sequential Vertex Coloring Algorithm does not always produce optimal coloring
Left: Proper coloring using Sequential Vertex Coloring algorithm requires at least 3 colors, because this algorithm assigns the color to 1, 2, 3, and then 4 sequentially.
Right: An optimal proper coloring of the graph requires only 2 colors. (Hence it is bipartite)

14.2.2 Applications

The Sequential Vertex Coloring algorithm has a variety of application and uses:

1. A graph is said to be *bipartite* if it can be colored using 2 colors. In particular, if the sequential vertex coloring requires at most 2 colors for a proper coloring of the graph, then the graph is said to be bipartite, but the reverse is not true. Even if the algorithm requires more than 2 colors, then also the graph may be bipartite, because the coloring produced by this algorithm is not optimal, as shown in Fig 14.2.2.
2. Geographic maps can be colored using this coloring algorithm. Also, according to the Four Color Theorem[8], not more than four colors are required for a proper coloring of the map (a loopless planar graph). Coloring of a sample geographic map is shown in Fig 14.2.3.
3. In Job Scheduling, so as to satisfy the constraints, all the clashing constraints (jobs which can't be executed simultaneously) can be represented in the form of a graph connected by an edge. Then, a proper coloring of the graph will solve this job scheduling problem, where colors represent the time slot to be assigned to a particular job. In fact, all such scheduling problems can be reduced to a graph coloring problem. e.g., Sudoku, Register Allocation, Frequency assignment to towers, etc.

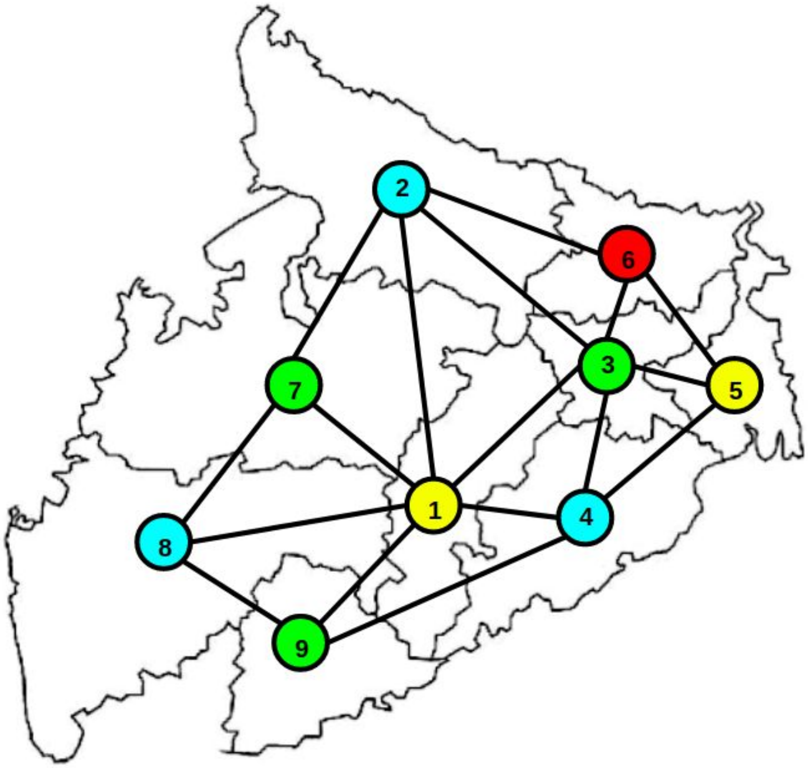


Fig 14.2.3: Coloring of a sample geographic map using sequential vertex coloring
At most 4 colors are required for a proper coloring of the given map.

14.2.3 Proposed File Structure

NOTE: The proposed file structure may change once the implementation begins.

Directory	Files
`doc/graphColoring/`	`CMakeLists.txt` `pgr_sequentialVertexColoring.rst`

`docqueries/graphColoring/`	`CMakeLists.txt`
	`doc-pgr_sequentialVertexColoring.result`
	`doc-pgr_sequentialVertexColoring.test.sql`
	`test.conf`

`include/graphColoring/`	`pgr_sequentialVertexColoring.hpp`
`include/drivers/graphColoring/`	`sequentialVertexColoring_driver.h`

`pgtap/graphColoring/sequentialVertexColoring/`	`sequentialVertexColoring-edge-cases.sql`
	`sequentialVertexColoring-innerQuery.sql`
	`sequentialVertexColoring-types-check.sql`
	`no_crash_test-sequentialVertexColoring.sql`

`sql/graphColoring/`	`CMakeLists.txt`
	`_sequentialVertexColoring.sql`
	`sequentialVertexColoring.sql`

`src/graphColoring/`	`CMakeLists.txt`
	`sequentialVertexColoring.c`
	`sequentialVertexColoring_driver.cpp`

14.2.4 Proposed Signature

NOTE: The proposed signatures and parameters may change once the implementation of this function begins.

The possible variants are:

- `pgr_sequentialVertexColoring()`

```
pgr_sequentialVertexColoring(edges_sql)
RETURNS SET OF (seq, node, color)
```

Parameters[10]

Parameter	Type	Description
edges_sql	`TEXT`	Inner SQL query, as described below.

Inner Query[10]

edges_sql: It should be 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.

Here,

`ANY-INTEGER` = SMALLINT, INTEGER, BIGINT
`ANY-NUMERICAL` = SMALLINT, INTEGER, BIGINT, REAL, FLOAT

Result Columns

Returns set of `(seq, node, color)`

Column	Type	Description
seq	`INT`	Sequential value starting from **1**.
node	`BIGINT`	Identifier of all the nodes in the graph.

color	BIGINT	Identifier of the color of the node. Ranges from 0 to the total number of nodes - 1.

14.2.5 Usage

Graph data is taken from pgRouting Sample Data.

```
SELECT id, source, target, cost, reverse_cost FROM edge_table;

 id | source | target | cost | reverse_cost
----+--------+--------+------+--------------
  1 |      1 |      2 |    1 |            1
  2 |      2 |      3 |   -1 |            1
  3 |      3 |      4 |   -1 |            1
  4 |      2 |      5 |    1 |            1
  5 |      3 |      6 |    1 |           -1
  6 |      7 |      8 |    1 |            1
  7 |      8 |      5 |    1 |            1
  8 |      5 |      6 |    1 |            1
  9 |      6 |      9 |    1 |            1
 10 |      5 |     10 |    1 |            1
 11 |      6 |     11 |    1 |           -1
 12 |     10 |     11 |    1 |           -1
 13 |     11 |     12 |    1 |           -1
 14 |     10 |     13 |    1 |            1
 15 |      9 |     12 |    1 |            1
 16 |      4 |      9 |    1 |            1
 17 |     14 |     15 |    1 |            1
 18 |     16 |     17 |    1 |            1
(18 rows)
```

Query

Find a proper coloring (sequential vertex coloring) of the given graph, i.e., assign colors to all the vertices of the graph such that no two vertices connected by an edge have the same color.

```
SELECT * FROM pgr_sequentialVertexColoring(
    ‘SELECT id, source, target, cost, reverse_cost FROM edge_table
     ORDER BY id’,
);
```

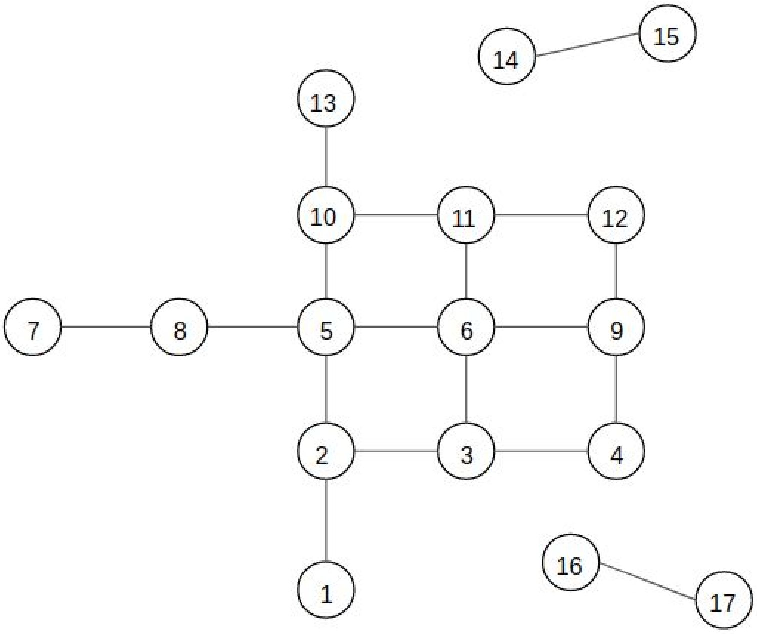


Fig 14.2.4: Representation of pgRouting Sample Data[7]

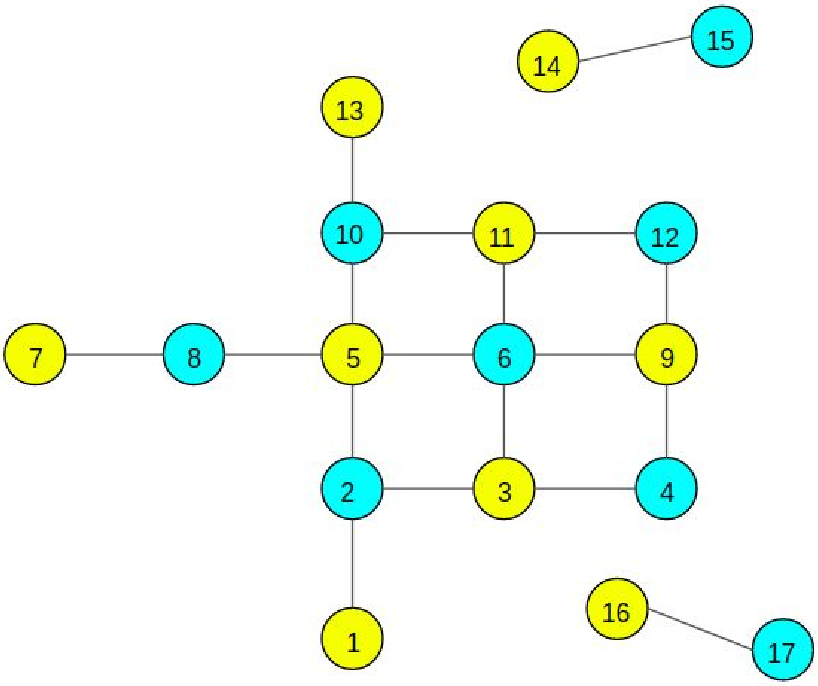


Fig 14.2.5: Final State of the given graph (Proper coloring).
Only two colors are required to color this graph. Hence it is bipartite.

Output

```
 seq | node | color
-----+------+-------
   1 |    1 |     0
   2 |    2 |     1
   3 |    3 |     0
   4 |    4 |     1
   5 |    5 |     0
   6 |    6 |     1
   7 |    7 |     0
   8 |    8 |     1
   9 |    9 |     0
  10 |   10 |     1
  11 |   11 |     0
  12 |   12 |     1
  13 |   13 |     0
  14 |   14 |     0
  15 |   15 |     1
  16 |   16 |     0
  17 |   17 |     1
(17 rows)
```

14.2.6 Visualization

Animation of the Sequential Vertex Coloring
https://github.com/krashish8/GSoC-pgRouting/issues/7#issuecomment-597593449
(External link because GIFs cannot be embedded in PDF format)

Color Value	Meaning
0	**Yellow**
1	**Blue**

14.3 Maximum Adjacency Search[11] (From Boost Graph Library[1])

Maximum Adjacency Search (MAS) or **Maximum Cardinality Search** is a vertex traversal algorithm that can be applied on an undirected graph, and the graph can be both weighted and unweighted.

The Maximum Adjacency Search algorithm maintains a set of vertices (say, a set *S*) of the graph at every step. Initially, S is empty and in each step, a vertex from the rest of the graph is shifted to S. The algorithm proceeds as follows: first any arbitrary vertex is selected from the graph and added to *S*. Then in every step, a vertex *v* which is not contained in *S* and is *most tightly connected* to *S* is added to *S*. Here, the most tightly connected vertex is that vertex *t* which is not in *S* and the sum of weights of the edges between *t* and *S* is highest among all other vertices not present in *S*. Ties (if any) are broken arbitrarily, i.e., in case many vertices exist which are tightly connected to S, then any one of them can be added to S.

Mathematically, the set S grows by adding a vertex `v ∉ S` to `S` such that,
`w(S, v) = max {w(S, u) ∀ u ∉ S},` where `w(S, u)` denotes the sum of weights of all the edges connecting `S` and `u`.

The algorithm continues adding vertices to S until all the vertices of the graph are traversed, i.e. it stops when S contains all the vertices of the graph.

This algorithm has been already implemented in the Boost Graph Library. The documentation can be found here, and the source code can be found here.

Fig 14.3.1 shows a graph in which maximum adjacency search is performed from vertex with id 4.
Fig 14.3.2 shows another state of the same graph. These figures demonstrate the vertices contained in the set S, and the next vertex which will be added to S, at the given state.

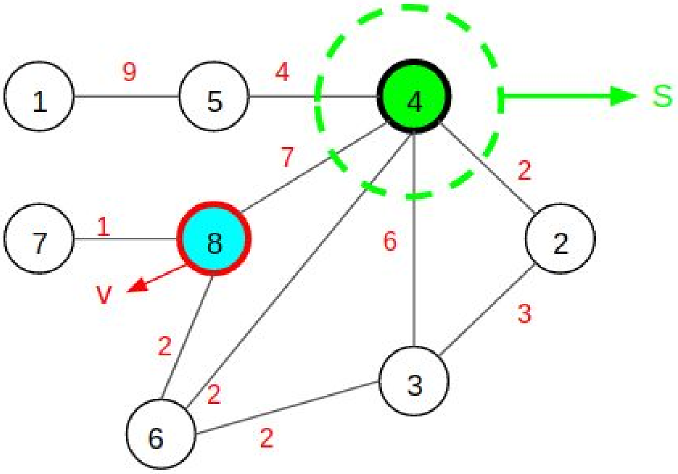


Fig 14.3.1: Let us perform a Maximum Adjacency Search of the graph, starting from vertex 4

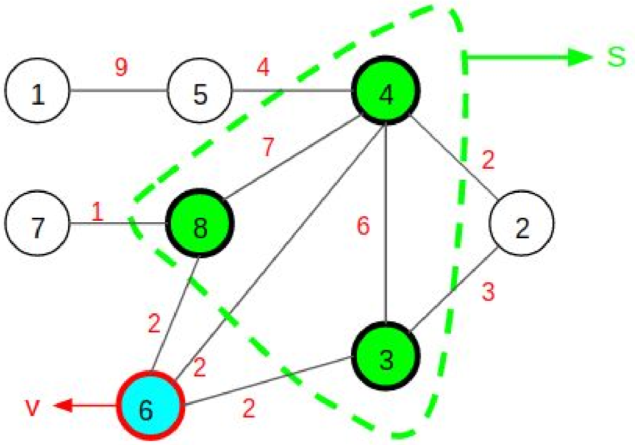


Fig 14.3.2: In this state of the graph, the vertices 4, 8 and 3 are visited in order and added to the set S. Now, out of the remaining vertices, the vertex 6 will be added next to the graph as it is most tightly connected to S. (sum of weights = 2 + 2 + 2 = 6)

14.3.1 Algorithm

```
// Maximum Adjacency Search from a start vertex s
procedure Maximum-Adjacency-Search(G, s)[11]:
      Declare max_priority_queue PQ[key, vertex]
      for each vertex u in V(G):
         PQ <-- push[0, u]            // push every vertex in PQ, key := 0
      update key[s] := 1 in PQ        // starting vertex
      while PQ is not empty:          // PQ contains unvisited vertices
         // Vertex u is visited now
         u := top element of PQ
         pop top element of PQ
         for each vertex v in Adj[u]:
                // Process edge E(u, v)
                if v is contained in PQ:
                       update key[v] := key[v] + weight[u, v] in PQ
         // Vertex u is finished visiting
```

This algorithm has a linear time complexity of `O(|V| + |E|)`, where |V| and |E| denote the total number of vertices and edges in the graph, respectively.

14.3.2 Applications

The Maximum Adjacency Search algorithm has many advanced applications. It can be used to:

- Find a global minimum cut of a graph. A global min-cut is a partition of the vertices of a graph into two subsets, such that the sum of weights of edges between the subsets is minimized. This method is used as a step in the Stoer-Wagner algorithm to calculate min-cut of a graph. To find global min-cut:

  - An arbitrary vertex is selected and Maximum Adjacency Search algorithm is applied on the graph, taking that vertex as the starting vertex.
  - The last vertex and second-last vertex traversed during the search algorithm are marked, and then the last vertex is merged to the second-last vertex, thereby shrinking the graph. Also, the weights of any edge from the two vertices to the remaining vertices are replaced by the sum of the weights of the previous two edges.
  - The search algorithm is applied repeatedly from the same source vertex and the graph is shrinked at every step. Also, the *cut-of-the-phase* is stored during each search traversal. Here, the *cut-of-the-phase* denotes the sum of the weights of the edges which must be removed to partition the graph into two subsets of vertices - the last vertex traversed in MAS algorithm and the rest vertices. Then finally, the minimum of all the cut-of-the-phase gives the min-cut of the graph.
  - Finding a min cut of the graph has several applications. It can be used in image segmentation or for detecting communities in social networks. It also shows the weakness of any graphical network.

- Find a perfect elimination ordering of the graph[13]. A perfect elimination ordering is the arrangement of the vertices of a graph such that for each vertex v, every vertex u which is a neighbor of v and occurs after v in order and including v itself, forms a clique (A clique or ‘complete subgraph’ is a subset of vertices in an undirected graph such that every two distinct vertices in the clique are adjacent). More formally, if $N(v_i)$ denotes the neighbors of vertex v_i with position in the ordering > i, then the set S containing v_i and $N(v_i)$ forms a clique.
- Compute minimal triangulations (chordal completion) in a graph[12], i.e., to compute triangulation with fewer numbers of edges. A triangulated graph (or chordal graph) is the one in which addition of any edge results in a non-planar graph. Also, a graph is chordal if and only if it has a perfect elimination ordering[13].

14.3.3 Proposed File Structure

NOTE: The proposed file structure may change once the implementation begins.

Directory	Files
`doc/mincut/`	`CMakeLists.txt` `pgr_maximumAdjacencySearch.rst`

`docqueries/mincut/`	`CMakeLists.txt`
	`doc-pgr_maximumAdjacencySearch.result`
	`doc-pgr_maximumAdjacencySearch.test.sql`
	`test.conf`

`include/mincut/`	`pgr_maximumAdjacencySearch.hpp`

include/drivers/mincut/	maximumAdjacencySearch_driver.h
pgtap/mincut/maximumAdjacencySearch/	maximumAdjacencySearch-edge-cases.sql maximumAdjacencySearch-innerQuery.sql maximumAdjacencySearch-types-check.sql no_crash_test-maximumAdjacencySearch.sql
sql/mincut/	CMakeLists.txt _maximumAdjacencySearch.sql maximumAdjacencySearch.sql
src/mincut/	CMakeLists.txt maximumAdjacencySearch.c maximumAdjacencySearch_driver.cpp

14.3.4 Proposed Signature

NOTE: The proposed signatures and parameters may change once the implementation of this function begins.

The possible variants are:

- `pgr_maximumAdjacencySearch()` Single vertex

```
pgr_maximumAdjacencySearch(edges_sql, start_vid)
RETURNS SET OF (seq, start_vid, node, edge, cost, agg_cost)
OR EMPTY SET
```

- `pgr_maximumAdjacencySearch()` Multiple vertices

```
pgr_maximumAdjacencySearch(edges_sql, start_vids)
RETURNS SET OF (seq, start_vid, node, edge, cost, agg_cost)
OR EMPTY SET
```

`EMPTY SET` is returned when no path exists or the root vertex is not present in the graph.

Parameters[10]

Parameter	Type	Description
edges_sql	`TEXT`	Inner SQL query, as described below.

start_vid	`BIGINT`	Identifier of the vertex of the graph from which traversal will start. ● Used on Single vertex.
start_vids	`ARRAY[ANY-INTEGER]`	Array of identifiers of the start vertices. ● Used on Multiple vertices. ● For optimization purposes, any duplicated value is ignored.

Inner Query[10]

edges_sql: It should be 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.

Here,

`ANY-INTEGER` = SMALLINT, INTEGER, BIGINT
`ANY-NUMERICAL` = SMALLINT, INTEGER, BIGINT, REAL, FLOAT

Result Columns

Returns set of `(seq, start_vid, node, edge, cost, agg_cost)`

Column	Type	Description
seq	`BIGINT`	Sequential value starting from **1**.
start_vid	`BIGINT`	Identifier of the root vertex. - In Multiple vertices, results are in ascending order.
node	`BIGINT`	Identifier of `node` reached using `edge`.
edge	`BIGINT`	Identifier of the edge used to arrive to `node`. - −1 when `node` = `start_vid`.
cost	`FLOAT`	Cost to traverse `edge`.
agg_cost	`FLOAT`	Aggregate cost from `start_vid` to `node`.

14.3.5 Usage

Graph data is taken from pgRouting Sample Data.

```
SELECT id, source, target, cost, reverse_cost FROM edge_table;

 id | source | target | cost | reverse_cost
----+--------+--------+------+--------------
  1 |      1 |      2 |    1 |            1
  2 |      2 |      3 |   -1 |            1
  3 |      3 |      4 |   -1 |            1
  4 |      2 |      5 |    1 |            1
  5 |      3 |      6 |    1 |           -1
  6 |      7 |      8 |    1 |            1
  7 |      8 |      5 |    1 |            1
  8 |      5 |      6 |    1 |            1
  9 |      6 |      9 |    1 |            1
 10 |      5 |     10 |    1 |            1
 11 |      6 |     11 |    1 |           -1
 12 |     10 |     11 |    1 |           -1
 13 |     11 |     12 |    1 |           -1
 14 |     10 |     13 |    1 |            1
 15 |      9 |     12 |    1 |            1
 16 |      4 |      9 |    1 |            1
 17 |     14 |     15 |    1 |            1
 18 |     16 |     17 |    1 |            1
(18 rows)
```

Query
Find the Maximum Adjacency Search of the graph with start vertex 1.

Note: In the below query, the last two rows are excluded from the pgRouting sample data because they denote two different connected components of the graph which do not include the vertex 1, and the Maximum Adjacency Search algorithm is generally effective only when the graph is connected.

```
SELECT * FROM pgr_maximumAdjacencySeaerch(
    ‘SELECT id, source, target, cost, reverse_cost FROM edge_table
     WHERE id < 17
     ORDER BY id’,
     1
);
```

Order of Traversal of vertices:
1 → 2 → 5 → 10 → 13 → 11 → 6 → 3 → 12 → 9 → 4 → 8 → 7

In this query, many times the case occurs that there exist several vertices which are tightly connected to the selected set of vertices. Here, in this implementation, the clashes are resolved by selecting that vertex which has a higher vertex id as compared to other vertices. The reason for choosing this is because of the implementation of algorithm using a max priority queue, the vertex with higher vertex id is given a higher priority over all other vertices which are also tightly connected.

Output

```
 seq | start_vid | node | edge | cost | agg_cost
-----+-----------+------+------+------+----------
   1 |         1 |    1 |   -1 |    0 |        0
   2 |         1 |    2 |    1 |    1 |        1
   3 |         1 |    5 |    4 |    1 |        2
   4 |         1 |   10 |   10 |    1 |        3
   5 |         1 |   13 |   14 |    1 |        4
   6 |         1 |   11 |   12 |    1 |        5
   7 |         1 |    6 |   11 |    1 |        6
   8 |         1 |    3 |    5 |    1 |        7
   9 |         1 |   12 |   13 |    1 |        8
  10 |         1 |    9 |   15 |    1 |        9
  11 |         1 |    4 |   16 |    1 |       10
  12 |         1 |    8 |    7 |    1 |       11
  13 |         1 |    7 |    6 |    1 |       12
(13 rows)
```

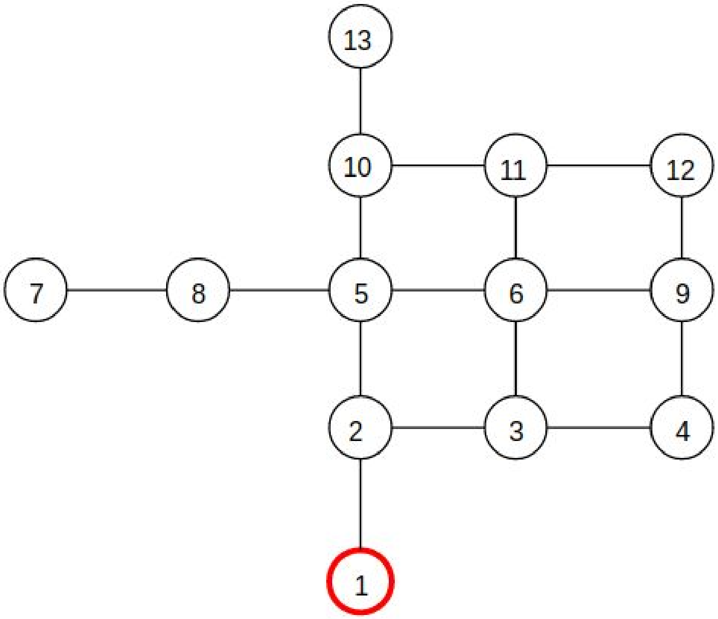


Fig 14.3.3: Representation of pgRouting Sample Data, excluding last 2 rows of the data[7]

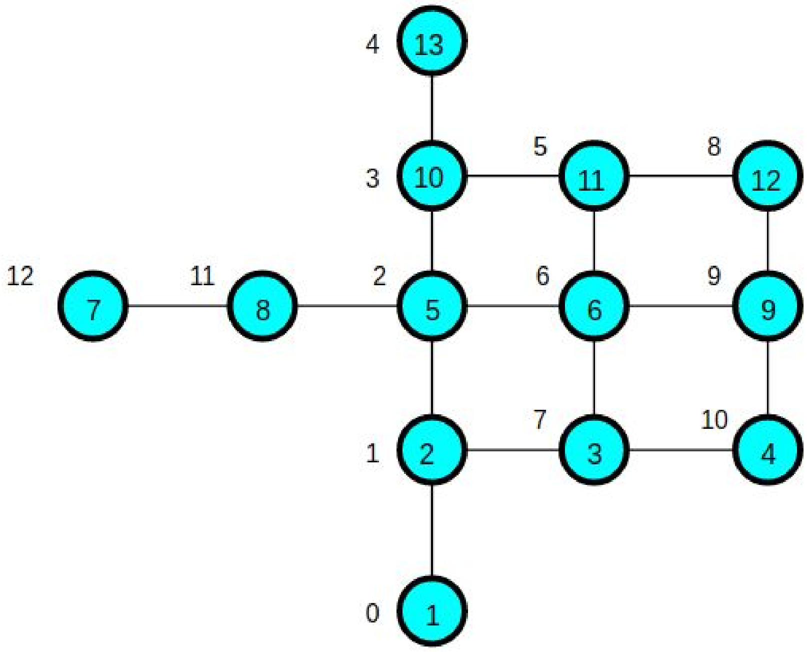


Fig 14.3.4: Final State of the given graph after Maximum Adjacency Search.
Here, the numbers beside the vertex denote their order of traversal.

Since, the vertex `7` is visited at the last, hence in the current Minimum Cut Phase, the graph can be split into two subset of vertices: `{7}` and `{1, 2, 3, 4, 5, 6, 8, 9, 10, 11, 12, 13}`. The sum of edges connecting these two subsets denotes the *cut-of-the-phase*. Here, it is the edge connecting vertex `7` and vertex `8` having weight `1`, and thus the *cut-of-the-phase* is `1`.

14.3.6 Visualization

Animation of the Maximum Adjacency Search
https://github.com/krashish8/GSoC-pgRouting/issues/7#issuecomment-599063745
(External link because GIFs cannot be embedded in PDF format)

Node / Edge Representation	Meaning
○	Unvisited vertex
○	Currently processing vertex
●	Visited and processed vertex

15. Future Work

I plan to finalize and complete all the algorithms mentioned in this proposal during the GSoC period. Also, if time permits, then I will implement the `pgr_maximumAdjacencySearch()` algorithm along with its tests and documentation.

There are other similar functions which can be implemented in the future:

1. `pgr_edgeColoring()` - it is similar to the `pgr_sequentialVertexColoring()` algorithm, except that it is applied to the edges of the graph instead of the vertices. In this, no two edges incident to a vertex can have the same color.

2. `pgr_KempeGraphColoring()` - Kempe's Graph Coloring algorithm[9] provides a method to color a planar loopless graph with at most 5 colors. It has several applications today, such as register allocation in the compilers. This can be implemented in the future.

16. References

[1]. The Boost Graph Library (BGL) - Version 1.72.0 Documentation.
[2]. Undirected DFS - Boost Graph Library (Boost 1.72.0 Library Documentation).
[3]. Depth First Search - Boost Graph Library (Boost 1.72.0 Library Documentation).
[4]. Depth First Search - Wikipedia, the Free Encyclopedia.
[5]. Sequential Vertex Coloring - Boost Graph Library (Boost 1.72.0 Library Documentation).
[6]. Graph Coloring - Wikipedia, the Free Encyclopedia.
[7]. pgRouting Sample Data - pgRouting v3.0.0-rc1.
[8]. Four Color Theorem - Wikipedia, the Free Encyclopedia.
[9]. Andrew W. Appel. *Kempe's Graph Coloring algorithm*. Princeton University; 2016.
[10]. pgr_kruskalDFS Documentation - pgRouting Manual v3.0.0-rc1.

[11]. Maximum Adjacency Search - Boost Graph Library (Boost 1.72.0 Library Documentation).
[12]. Anne Berry, Jean R. S. Blair, and Pinar Heggernes. *Maximum Cardinality Search for Computing Minimal Triangulations*.
[13]. Jean R. S. Blair, Barry W. Peyton. *An introduction to Chordal Graphs and Clique Trees*. Oak Ridge National Laboratory; November 1992. p. 4-9.

17. Resume

Ashish Kumar

B. Tech, 2nd Year
Computer Science & Engineering
IIT (BHU) Varanasi, India

Mobile: +91-6205144592
Email: ashishkumar.cse18@iitbhu.ac.in
Github: https://github.com/krashish8
Linkedin: https://linkedin.com/in/ashishkr23438

Education

- **Indian Institute of Technology (BHU), Varanasi** *Varanasi, India.*
  B. Tech. in Computer Science & Engineering (Current CPI: **9.83 / 10.00**) *2018 - 2022*

Semester	**I**	**II**	**III**
SPI	9.94 / 10.00	9.76 / 10.00	9.81 / 10.00

- **Subhash Public School** *Giridih, India.*
  Class XII, Central Board of Secondary Education (CBSE); **92.00%** *2016 - 2018*

- **Carmel School** *Giridih, India.*
  Class X, Indian Certificate of Secondary Education (ICSE); **96.00%** *2005 - 2016*

Skills and Interests

- **Languages:** C, C++, Python, JavaScript, Bash, PHP, Java
- **Databases:** PostgreSQL / PostGIS, MySQL.
- **Technologies/Tools:** Bootstrap, Django, Django REST Framework, Git, Angular, Vue.js.
- **Interests:** Data Structures & Algorithms, Problem Solving, Software Development, Web development, Machine Learning.

Projects and Technical Experience

Let's Meet - Slack Bot + Web App (Hackathon) *Feb, 2020*

- A Slack bot and a Web App with a separate backend. (Won us 3rd Prize in Hackathon)
- Exposure: Python, Django, Django REST Framework, NodeJS. (Source Codes)

Swiftrail - Railway Management System *Aug, 2019 – Dec, 2019*

- Database management web application which helps in Railway Ticket booking and Enquiry System.
- Exposure: Python, Django, MySQL, HTML, CSS, Bootstrap (Source Code)

Tech Team, Technex ’20 *Sep, 2019 – Dec, 2019*

- Worked as a member of the backend team in making Django REST API for the Technex ’20 website.
- Exposure: Python, Django REST Framework, Angular. (Backend)

Tech Team, Spardha ’19 *May, 2019 – July, 2019*

- Made both frontend and backend of the Spardha ’19 website.
- Exposure: HTML, CSS, Bootstrap, PHP, MySQL. (Source Code)

C++ Public Projects *March 2019 – Current*

- GitHub repositories containing my submitted C++ Online Judge solutions and C++ algorithm templates used by me frequently.
- GitHub repositories containing my codes submitted during the Data Structures and Algorithms course, Algorithms course and Operating System course at IIT (BHU) Varanasi.

Relevant Courses

- Basic Programming in C / C++
- Data Structures and Algorithms
- Algorithms
- Operating System
- Graph Theory

Achievements

- **ICPC 2019:** Secured All India Team Rank 126 in Asia Amritapuri Regional Contest. *(Onsite Round)*
- **Competitive Programming:** An Expert level programmer on Codeforces (Profile) and Division-1 (5 star) programmer on Codechef (Profile).
- **Open Source:** An enthusiastic Open Source contributor (Profile).
- **IIT (BHU) Varanasi:** Institute Topper in 1st Semester, securing 9.94 / 10.00 SPI. *(2018)*
- **JEE (Main) 2018:** Secured All India Rank 1295 amongst 1.2 million aspirants.
- **JEE (Advanced) 2018:** Secured All India Rank 5853 amongst 0.2 million aspirants.
- **Class X (ICSE):** District Topper in Class X ICSE Board. *(2016)*
- **Club of Programmers:** Secured 3rd position in the Summer Development Event *(2019)*.
- **Codefest ‘19 IIT (BHU) Varanasi:** Ranked 1st among sophomores in CTF (InfoSec contest) *(2019)*.
- **Club of Programmers:** An active member of the Club of Programmers, IIT (BHU) Varanasi.