Implementation of a framework which supports addition of contraction techniques for pgRouting

Index

Contact Details

Name: Sankepally Rohith Reddy
Country: India
Email: rohithreddy2219@gmail.com
Phone: +91 9652974344

Title

Implementation of a framework which supports addition of contraction techniques for pgRouting.

Synopsis

Contraction is a technique used to speed up shortest-path computation by first creating contracted versions of the network. There are different ways of contracting a network. I am proposing a framework which supports the addition of new contraction techniques, and two contraction algorithms: dead end contraction and linear contraction.

Installation of the framework together with the dead end contraction and also the implementation of linear contraction which will be used to refine the framework is intended to be completed during the GSoC.

Benefits to Community

Contracting a graph becomes a crucial operation when talking about big graphs like the graphs involved in routing across cities, countries, continents or the whole world. Large businesses which have a problem of routing through large road networks can be solved by using contraction techniques. This idea, if and when implemented in pgRouting can help various organisations around the world to optimize their route calculation over larger road networks. Contraction can also be combined with many other route planning techniques, leading to improved performance for many-to-many routing and goal directed routing.

Deliverables

The deliverables would be:

- A framework which supports the addition of new contraction algorithms.
- Implementation of Dead end contraction
- Implementation of Linear contraction which will be used to refine the framework.
- Proper documentation and tests for the above-mentioned components.

Timeline

Before the official coding time:

- Make a wiki page for setting up TODO lists and weekly reports.
- Getting familiar with the community, the version control system, the documentation and test system used.
- Make a repository on github for upload and access of source code.
- Getting familiar with postgis and understanding pgrouting architecture.
- Get more familiar with PL/pgSQL Procedural Language.
- Try to look at other projects which use contraction hierarchies and try to understand, how they implement it.

Week 1 (Official coding period starts):

- Analyse and design the data structures which are used in contracting the network.
- Design the schema for storing the contracted graph.

Week 2:

- Design a generic template model for contraction, which supports the addition of new contraction techniques.

Week 3:

- Implement the basic version of the template model of contraction.
- Design an algorithm to implement dead end contraction.

Week 4 to 6

- Implementation of the dead end contraction.

Week 7:

- Design an algorithm to implement linear contraction.

Week 8 to 11:

- Refine the framework by implementing linear contraction.
- Rigorous testing and bug fixes.

Week 12:

- Documentation for the whole project

Weekly Report Format :

- Work done in the week.
- Problems faced during the work.
- Work to be done next week.

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

Yes, I completely understand and am aware of my commitment levels. I am fully prepared for the work and will put in my best efforts.

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

I do not have any conflicts during the official coding period.

Related Work

Contraction algorithms are implemented and used in many organisations like Open Source Routing Machine(http://wiki.openstreetmap.org/wiki/Open_Source_Routing_Machine), CartoType(http://www.cartotype.com/index.html) and GraphHopper(http://wiki.openstreetmap.org/wiki/GraphHopper) . They work on specific formats

of data and the contraction algorithms depend on the type of data. In this proposal, the contraction algorithms are generic and do not depend on the type of data. In addition to this a contraction framework is also being implemented which is extendible and allows the addition of new contraction algorithms.

Biographical Information

I am 3rd year computer science undergraduate from International Institute of Information Technology(IIIT-H) in Hyderabad, India. I am pursuing my Bachelors in Computer Science and Masters by research in Spatial Informatics. GIS is my specialisation for research in my Masters. I am looking forward to pursue a career in the fields with GIS as a core concept. I am interested in open source development as it is extremely helpful to developers everywhere to create new and improved programs to solve real world problems.

I have been working on contraction hierarchies as a part of my research in the field of Spatial Informatics and started learning about the different ways of contracting a graph. After that some ideas came up, and then wanted to create a software model regarding the same. Dead end contraction was implemented as open source in https://github.com/sankepallyrohithreddy/OSMContraction , and then wanted to have my code integrated into pgrouting, so I asked pgrouting team for help. As part of the instructions given to me unit tests were made and then after making the first unit test it was noticed that the results were wrong. After discussing my idea with the pgrouting team, their feedback helped me a lot in refining and redesigning my idea.

Studies

What is your School and degree?

School : International Institute of Information Technology (IIIT-H) in India.
Degree: Bachelors in Computer Science and Masters by research in Spatial Informatics.

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

Yes, this application will contribute much to my ongoing studies. This project will be an added asset to my ongoing degree and would help me gain a better understanding on how GIS softwares work. I will get hands-on experience on contraction techniques and pgRouting, and the experience and information I will obtain in the course of developing this application will contribute a lot to my Master's thesis.

Programming and GIS

Computing experience

Languages : C, C++, python, bash, java, javascript.
APIs : Google maps API, Android API.
Operating Systems : Linux(Ubuntu 14.04), Windows.
Server side scripting : PHP , Node Js
Networking : Socket Programming

GIS Experience

I am quite used to various GIS related softwares like pgRouting, QGIS, MapBox, Osm2pgrouting and OpenstreetMaps.

GIS programming

Since GIS is my specialisation for research in my Masters, I have been using multiple GIS softwares as mentioned above.

I have done a project which is aims at navigating a user inside a university campus. The aim of this project was to provide information about some important locations inside the university and also route the user between any two points inside the university. Routing was implemented using pgRouting, by using the OpenstreetMap data of my university. Osm2pgrouting tool was used to import the OSM data into the pgRouting database.The other softwares used in this project were QGIS and MapBox.

GSoC participation

This is the first time I am participating in the GSoC program. I did not submit a proposal to any other organisation.

Proposal: Contraction Algorithm

Introduction

Contracting a graph becomes a crucial operation when talking about big graphs like the graphs involved in routing across cities, countries, continents or the whole world.

The contraction level and contraction operations can become very complex, as the complexity of the graphs grows.
For this proposal, we are making our contraction algorithm as simple as possible so that more contraction operations can be added in the future.
We are not aiming with this work to implement all the possible contraction operations but to give a framework such that adding a contraction operation can be easily achieved.
For this contraction proposal, I am only making 2 operations:

1. **Dead end contraction**: vertices have one incoming edge
2. **Linear contraction**: vertices have one incoming and one outgoing edge

And with the additional characteristics:

- The user can forbid to contract a particular set of nodes or edges.
- The user can decide how many times the cycle can be done.
- If possible, the user can also decide the order of the operations on a cycle.

The contraction skeleton

In general we have an initial set up that may involve analysing the graph given as input and setting the non contractible nodes or edges. We have a cycle that will go and perform a contraction operation until while possible, and then move to the next contraction operation.
The framework is implemented such that it is extendible and supports the addition and removal of a contraction operation. Adding a new operation then becomes an “easy” task; more things might be involved, because the characteristics of the graph change each time it is contracted, so some interaction between contractions has to be implemented also.

Procedure

For contracting, we are going to cycle as follows

```
input: G(V,E);
removed_vertices = {};

<initial setup>
do N times {

    while ( <conditions for 1> ) {
        < contraction operation 1 >
    }

    while ( <conditions for 2> ) {
        < contraction operation 2>
    }
    .....
}
output: G'(V',E'), removed_vertices
```

Contraction operations for this implementation

Dead end contraction

Characteristics:

V1: set of vertices with 1 incoming edge in increasing order of id:
Edges with the same identifier are considered the same edge and if it has the reverse_cost valid the outgoing edge is ignored

```
while ( V1 is not empty ) {

    delete vertex of V1
    the deleted vertex add it to removed_vertices
    vertex that leads to removed vertex, inherits the removed vertex

    <adjust any conditions that might affect other contraction operation>
}
```

Linear contraction

Characteristics:

V2: vertex with 1 incoming edge and 1 outgoing edge:
The outgoing edge must have different identifier of the incoming edge

```
while ( V2 is not empty ) {

    delete vertex of V2
    create edge (shortcut)
    the deleted vertex add it to removed_vertices
    newly created edge, inherits the removed vertex

    <adjust any conditions that might affect other contraction operations>
}
```

Notation

V: is the set of vertices
E: is the set of edges
G: is the graph
V1: is the set of *dead end* vertices
V2: is the set of *linear* vertices
removed_vertices: is the set of removed vertices

The contracted graph will be represented with two parameters, the modified Graph, and the removed_vertices set.
removed_vertices = {(v, 1):{2}, (e, 20):{3}}.

The above notation indicates:

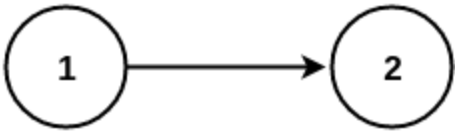
Vertex 2 is removed, and belongs to vertex 1 subgraph
Vertex 3 is removed, and belongs to edge 20 subgraph

Examples

For simplicity all the edges in the examples have unit weight.

Dead End

Perform dead end contraction operation first and then linear contraction
1 cycle of contraction.



Input: `G = {V:{1, 2}, E:{(1, 2)}}`

Initial Setup:

```
removed_vertices={}
V1 = {2}
V2 = {}
```

Procedure:

```
V1 = {2} is not empty
    V1 = {}
    V2 = {}
    G = {V:{1}, E:{}}
    removed_vertices = {(v, 1):{2}}.
V1 is empty
```

Since V1 is empty we go on to the next contraction operation

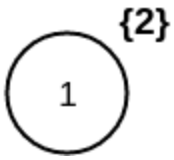
```
V2 is empty
```

So we do not perform any linear contraction operation.

Results:

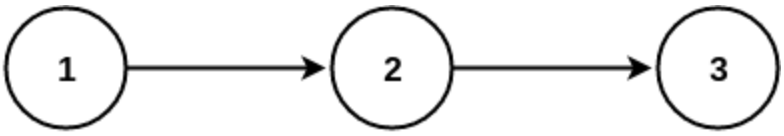
```
G = {V:{1}, E:{}}
removed_vertices = {(v, 1):{2}}
```

Visually the results are



Linear contraction

Perform linear contraction operation first and then dead end contraction
1 cycle of contraction.



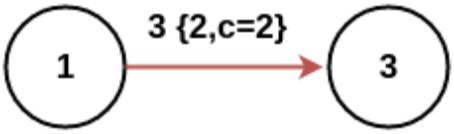
Input: G = {V:{1, 2, 3}, E:{1(1, 2), 2(2, 3)}}

Initial Setup:

```
removed_vertices={}
V1 = {3}
V2 = {2}
```

Procedure:

```
V2 = {2} is not empty

    V1 = {3}
    removed_vertices = {(e, 3):{2}}
    V2 = {}

   G = {V:{1, 3}, E:{3(1,3,c=2)}}
V2 is empty
```



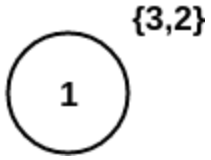
Since V2 is empty we go on to the next contraction operation

```
V1 = {3} is not empty

    V1 = {}
    V2 = {}
    removed_vertices = {(v, 1):{3, 2}}.
    G = {V:{1}, E:{}}

V1 is empty
```

Results:

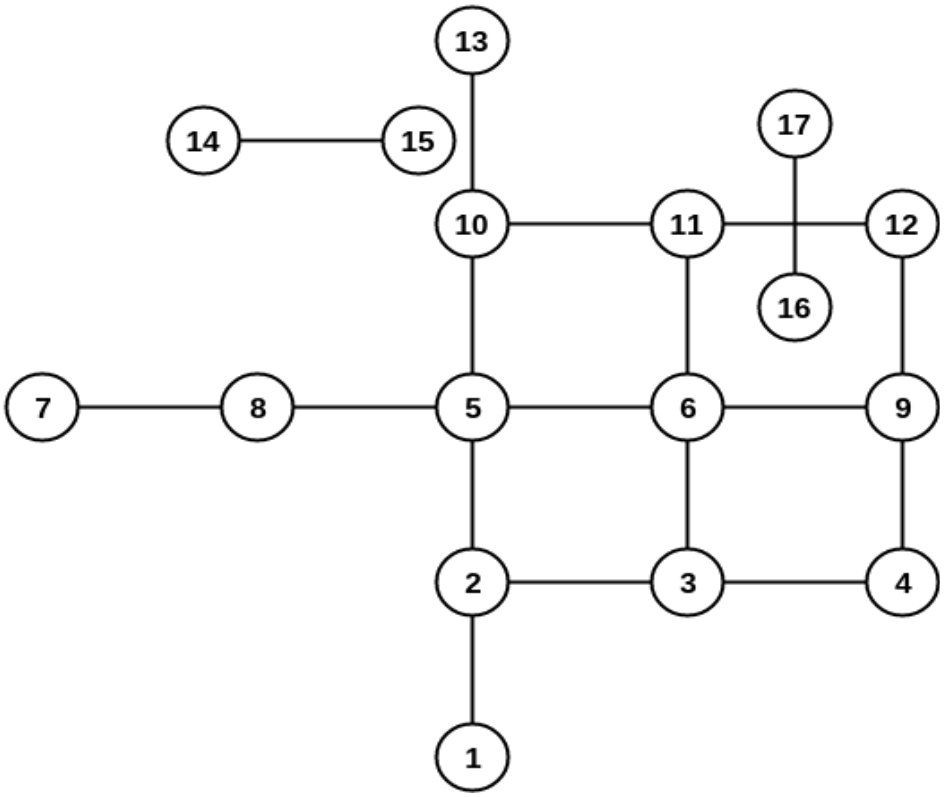
```
removed_vertices = {(v, 1):{3, 2}}.
G = {V:{1}, E:{}}
```

Visually the results are



Sample Data

Perform dead end contraction operation first and then linear contraction
1 cycle of contraction.



Input: `G = {V:{1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17}, E:{1(1, 2), 2(2,3), 3(3,4), 4(2,5), 5(3,6), 6(7,8), 7(8,5), 8(5,6), 9(6,9), 10(5,10), 11(6,11), 12(10,11), 13(11,12), 14(10,13), 15(9,12), 16(4,9), 17(14,15), 18(16,17)}}`

Initial Setup:

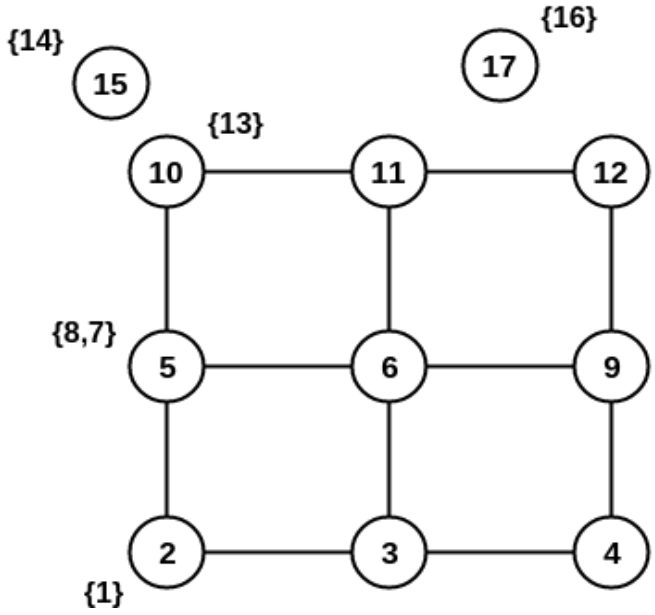
```
removed_vertices={}
V1 = {1, 7, 13, 14, 15, 16, 17}
V2 = {4, 8, 12}
```

Procedure:

```
V1 = {1, 7, 13, 14, 15, 16, 17} is not empty
    V1 = {7, 13, 14, 15, 16, 17}
    V2 = {2, 4, 8, 12}
    G = {V:{2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17},
    E:{2(2, 3), 3(3, 4), 4(2, 5), 5(3, 6), 6(7, 8), 7(8, 5), 8(5, 6), 9(6, 9),
10(5, 10), 11(6, 11), 12(10, 11), 13(11, 12), 14(10, 13), 15(9, 12), 16(4, 9),
17(14, 15), 18(16, 17)}}
    removed_vertices = {(v, 2):{1}}.

V1 = {7, 13, 14, 15, 16, 17} is not empty
    V1 = {8, 13, 14, 15, 16, 17}
    V2 = {2, 4, 12}
    G = {V:{2, 3, 4, 5, 6, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17},
    E:{2(2, 3), 3(3, 4), 4(2, 5), 5(3, 6), 7(8, 5), 8(5, 6), 9(6, 9), 10(5,
10), 11(6, 11), 12(10, 11), 13(11, 12), 14(10, 13), 15(9, 12), 16(4, 9), 17(14,
15), 18(16, 17)}}
    removed_vertices = {(v, 2):{1}, (v, 8):{7}}.

V1 = {8, 13, 14, 15, 16, 17} is not empty
    V1 = {13, 14, 15, 16, 17}
    V2 = {2, 4, 12}
    G = {V:{2, 3, 4, 5, 6, 9, 10, 11, 12, 13, 14, 15, 16, 17},
    E:{2(2, 3), 3(3, 4), 4(2, 5), 5(3, 6), 8(5, 6), 9(6, 9), 10(5, 10), 11(6,
11), 12(10, 11), 13(11, 12), 14(10, 13), 15(9, 12), 16(4, 9), 17(14, 15),
18(16, 17)}}
    removed_vertices = {(v, 2):{1}, (v, 5):{8,7}}.

V1 = {13, 14, 15, 16, 17} is not empty

    V1 = {14, 15, 16, 17}
    V2 = {2, 4, 10, 12}
    G = {V:{2, 3, 4, 5, 6, 9, 10, 11, 12, 14, 15, 16, 17},
    E:{2(2, 3), 3(3, 4), 4(2, 5), 5(3, 6), 8(5, 6), 9(6, 9), 10(5, 10), 11(6,
11), 12(10, 11), 13(11, 12), 15(9, 12), 16(4, 9), 17(14, 15), 18(16, 17)}
removed_vertices = {(v, 2):{1}, (v, 5):{8,7}, (v, 10):{13}}.


V1 = {14, 15, 16, 17} is not empty
    V1 = {16, 17}
    V2 = {2, 4, 10, 12}
    G = {V:{2, 3, 4, 5, 6, 9, 10, 11, 12, 15, 16, 17},
    E:{2(2, 3), 3(3, 4), 4(2, 5), 5(3, 6), 8(5, 6), 9(6, 9), 10(5, 10), 11(6,
11), 12(10, 11), 13(11, 12), 15(9, 12), 16(4, 9)}, 18(16, 17)}
removed_vertices = {(v, 2):{1}, (v, 5):{8, 7}, (v, 10):{13}, (v, 15):{14}}.
```

V1 = {16, 17} is not empty
    V1 = {}
    V2 = {2, 4, 10, 12}
    G = {V:{2, 3, 4, 5, 6, 9, 10, 11, 12, 15, 17},
    E:{2(2, 3), 3(3, 4), 4(2, 5), 5(3, 6), 8(5, 6), 9(6, 9), 10(5, 10), 11(6, 11), 12(10, 11), 13(11, 12), 15(9, 12), 16(4, 9)}}
removed_vertices = {(v, 2):{1}, (v, 5):{8,7}, (v, 10):{13}, (v, 15):{14}, (v, 17):{16}}.

Since V1 is empty we go on to the next contraction operation



V2 = {2, 4, 10, 12} is not empty
    V1 = {}
    V2 = {4, 10, 12}
    G = {V:{3, 4, 5, 6, 9, 10, 11, 12, 15, 17},
    E:{19(3, 5, c=2), 3(3, 4), 5(3, 6), 8(5, 6), 9(6, 9), 10(5, 10), 11(6, 11), 12(10, 11), 13(11, 12), 15(9, 12), 16(4, 9)}}
    removed_vertices = {(e, 19):{1, 2}, (v, 2):{1}, (v, 5):{8, 7}, (v, 10):{13}, (v, 15):{14}, (v, 17):{16}}.

V2 = {4, 10, 12} is not empty
    V1 = {}
    V2 = {10, 12}
    G = {V:{3, 5, 6, 9, 10, 11, 12, 15, 17},

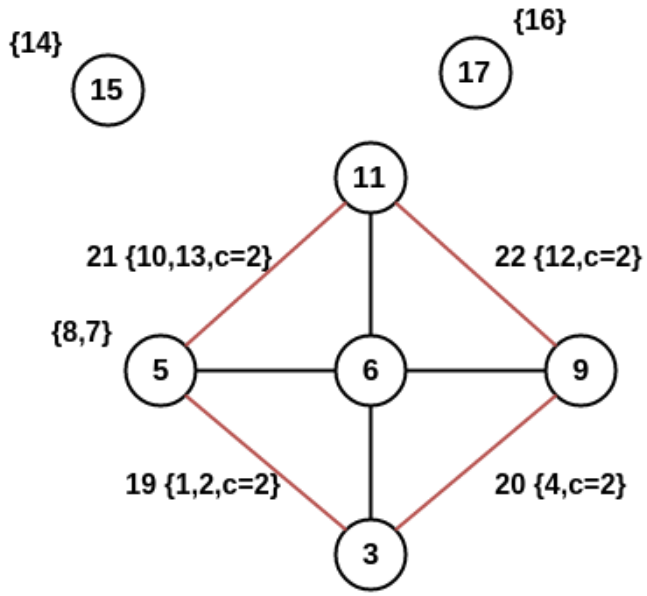
```
    E:{19(3, 5, c=2),20(3, 9, c=2), 5(3, 6), 8(5, 6), 9(6, 9), 10(5, 10), 11(6,
11), 12(10, 11), 13(11, 12), 15(9, 12)}}
    removed_vertices = {(e, 19):{1, 2}, (e, 20):{4}, (v, 2):{1}, (v, 5):{8, 7},
(v, 10):{13}, (v, 15):{14}, (v, 17):{16}}.

V2 = {10, 12} is not empty
    V1 = {}
    V2 = {12}
    G = {V:{3, 5, 6, 9, 11, 12, 15, 17},
    E:{19(3, 5, c=2),20(3, 9, c=2), 21(5, 11, c=2), 5(3, 6), 8(5, 6), 9(6, 9),
11(6, 11), 13(11, 12), 15(9, 12)}}
    removed_vertices = {(e, 19):{1, 2}, (e, 20):{4}, (e, 21):{10, 13}, (v,
2):{1}, (v, 5):{8, 7}, (v, 15):{14}, (v, 17):{16}}.

V2 = {12} is not empty
    V1 = {}
    V2 = {}
    G = {V:{3, 5, 6, 9, 11, 15, 17},
    E:{19(3, 5, c=2), 20(3, 9, c=2), 21(5, 11, c=2), 22(9, 11, c=2), 5(3,6),
8(5,6), 9(6,9), 11(6,11)}}
    removed_vertices = {(e, 19):{1,2}, (e, 20):{4}, (e, 21):{10,13}, (e,
22):{12}, (v, 2):{1}, (v, 5):{8, 7}, (v, 15):{14}, (v, 17):{16}}.
```

Since V1 and V2 are empty we stop our contraction here.

Results:

```
G = {V:{3, 5, 6, 9, 11, 15, 17},
E:{19(3, 5, c=2) , 20(3, 9, c=2), 21(5, 11, c=2), 22(9, 11, c=2), 5(3, 6), 8(5,
6), 9(6, 9), 11(6, 11)}}
removed_vertices = {(e, 19):{1, 2}, (e, 20):{4}, (e, 21):{10, 13}, (e,
22):{12}, (v, 2):{1}, (v, 5):{8, 7}, (v, 15):{14}, (v, 17):{16}}.
```

Visually the results are



Proof of Concept

This proof of concept demonstrates how contraction would reduce the computation to calculate the shortest path using dijkstra.

Let us take the example of the sample data which is contracted as shown above. We have the following data:

Original Graph 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)
```

Addition of a new column

Adding a new column which tells us whether the edge is a new edge added by the algorithm.

```
ALTER TABLE edge_table ADD is_contracted BOOLEAN DEFAULT false;
```

Addition of new edges by the algorithm

We are going to add manually the new edges of the contracted graph

```
INSERT INTO edge_table(source, target, cost, reverse_cost, is_contracted)
VALUES (3, 5, 2, 2, true);
INSERT INTO edge_table(source, target, cost, reverse_cost, is_contracted)
VALUES (3, 9, 2, 2, true);
INSERT INTO edge_table(source, target, cost, reverse_cost, is_contracted)
VALUES (5, 11, 2, 2, true);
INSERT INTO edge_table(source, target, cost, reverse_cost, is_contracted)
VALUES (9, 11, 2, 2, true);
```

Contracted Graph Data

As shown in the above figure, the vertices of the contracted graph are {3, 5, 6, 9, 11, 15, 17}. So to represent the contracted graph, we choose only those edges which have their source in {3, 5, 6, 9, 11, 15, 17} and their target in {3, 5, 6, 9, 11, 15, 17}.

```
SELECT id, source, target, cost, reverse_cost, is_contracted FROM
edge_table where source IN (3, 5, 6, 9, 11, 15, 17) AND target IN (3,
5, 6, 9, 11, 15, 17);
```

```
 id | source | target | cost | reverse_cost | is_contracted
----+--------+--------+------+--------------+---------------
  5 |      3 |      6 |    1 |           -1 | f
  8 |      5 |      6 |    1 |            1 | f
  9 |      6 |      9 |    1 |            1 | f
 11 |      6 |     11 |    1 |           -1 | f
 19 |      3 |      5 |    2 |            2 | t
 20 |      3 |      9 |    2 |            2 | t
 21 |      5 |     11 |    2 |            2 | t
 22 |      9 |     11 |    2 |            2 | t
(8 rows)
```

We perform shortest path computation on the original graph with some selected vertices and edges, which are chosen based on the contracted graph.

There are five cases which arise when calculating the shortest path between a given source and target,

- **Case 1:** Both source and target belong to the contracted graph.
- **Case 2:** Source belongs to a contracted graph, while target belongs to a vertex subgraph.
- **Case 3:** Source belongs to a contracted graph, while target belongs to an edge subgraph.
- **Case 4:** Source belongs to a vertex subgraph, while target belongs to an edge subgraph.
- **Case 5:** The path contains a new edge added by the contraction algorithm.

Examples

Case 1: Both source and target belong to the contracted graph.

Routing from 3 to 11

Since 3 and 11 both are in the contracted graph it is not necessary expand the graph.

```
SELECT * FROM pgr_dijkstra(
    'SELECT id, source, target, cost, reverse_cost
     FROM edge_table
     WHERE source IN (3, 5, 6, 9, 11, 15, 17)
           AND target IN (3, 5, 6, 9, 11, 15, 17)',
    3, 11, false);

    seq  | path_seq | node | edge | cost | agg_cost
    -----+----------+------+------+------+----------
       1 |        1 |    3 |    5 |    1 |        0
       2 |        2 |    6 |   11 |    1 |        1
       3 |        3 |   11 |   -1 |    0 |        2


(3 rows)
```

Case 2: Source belongs to a contracted graph, while target belongs to a vertex subgraph.

Routing from 3 to 7

Since 7 is in the contracted subgraph of vertex 5, it is necessary to expand that vertex by adding {7, 8} to the vertex set, so the vertex set becomes {3, 5, 6, 9, 11, 15, 17 , 7, 8}

```
SELECT  *  FROM  pgr_dijkstra(
    'SELECT id, source, target, cost, reverse_cost
     FROM edge_table
     WHERE source IN (3, 5, 6, 9, 11, 15, 17,   7, 8)
           AND target IN (3, 5, 6, 9, 11, 15, 17,   7, 8)',
    3, 7, false);


     seq | path_seq | node | edge | cost | agg_cost
    -----+----------+------+------+------+----------
       1 |        1 |    3 |   19 |    2 |        0
       2 |        2 |    5 |    7 |    1 |        2
       3 |        3 |    8 |    6 |    1 |        3
       4 |        4 |    7 |   -1 |    0 |        4

(4 rows)
```

Case 3: Source belongs to a contracted graph, while target belongs to an edge subgraph.

Routing from 3 to 13

Since 13 is in the contracted subgraph of edge (5, 11), it is necessary to expand that edge by adding {10, 13} to the vertex set, so the vertex set becomes {3, 5, 6, 9, 10, 11, 13, 15, 17}

```
SELECT * FROM pgr_dijkstra(
    'SELECT id, source, target, cost, reverse_cost
     FROM edge_table
     WHERE source IN (3, 5, 6, 9, 11, 15, 17,    10, 13)
           AND target IN (3, 5, 6, 9, 11, 15, 17,    10, 13)',
    3, 13, false);
```

```
 seq | path_seq | node | edge | cost | agg_cost
-----+----------+------+------+------+----------
   1 |        1 |    3 |    5 |    1 |        0
   2 |        2 |    6 |   11 |    1 |        1
   3 |        3 |   11 |   12 |    1 |        2
   4 |        4 |   10 |   14 |    1 |        3
   5 |        5 |   13 |   -1 |    0 |        4
(5 rows)
```

Case 4: Source belongs to a vertex subgraph, while target belongs to an edge subgraph.

Routing from 7 to 13

Since 13 is in the contracted subgraph of edge (5, 11), it is necessary to expand that edge by adding {10, 13} to the vertex set, and since 7 is in the contracted subgraph of vertex 5, it is necessary to expand that vertex by adding {7, 8} vertex set, so the vertex set becomes (3, 5, 6, 7, 8, 9, 10, 11, 13, 15, 17)

```
SELECT * FROM pgr_dijkstra(
    'SELECT id, source, target, cost, reverse_cost
     FROM edge_table
     WHERE source IN (3, 5, 6, 9, 11, 15, 17,    7, 8,   10, 13)
           AND target IN (3, 5, 6, 9, 11, 15, 17,    7, 8,   10, 13)',
    7, 13, false);
```

```
 seq | path_seq | node | edge | cost | agg_cost
-----+----------+------+------+------+----------
   1 |        1 |    7 |    6 |    1 |        0
   2 |        2 |    8 |    7 |    1 |        1
   3 |        3 |    5 |   10 |    1 |        2
   4 |        4 |   10 |   14 |    1 |        3
   5 |        5 |   13 |   -1 |    0 |        4
(5 rows)
```

Case 5: The path contains a shortcut.

Routing from 3 to 9

Since 3 and 9 both are in the contracted graph it is not necessary expand the graph.

```
SELECT * FROM pgr_dijkstra(
    'SELECT id, source, target, cost, reverse_cost
     FROM edge_table
     WHERE source IN (3, 5, 6, 9, 11, 15, 17)
           AND target IN (3, 5, 6, 9, 11, 15, 17)',
    3, 9, false);
```

```
 seq | path_seq | node | edge | cost | agg_cost
-----+----------+------+------+------+----------
   1 |        1 |    3 |   20 |    2 |        0
   2 |        2 |    9 |   -1 |    0 |        2
(2 rows)
```

```
SELECT is_contracted FROM edge_table WHERE id = 20;
```

```
 is_contracted
---------------
 t
(1 row)
```

This implies that it is a shortcut and should be expanded. The contracted subgraph of edge 20(3, 9, c=2) is {4}. It is necessary to expand the edge by adding {4} to the vertex set, so the vertex set becomes {3, 4, 5, 6, 9, 11, 15, 17}.

```
SELECT * FROM pgr_dijkstra(
```

```
    'SELECT id, source, target, cost, reverse_cost
     FROM edge_table
     WHERE source IN (3, 5, 6, 9, 11, 15, 17,  4)
           AND target IN (3, 5, 6, 9, 11, 15, 17,  4)
           AND is_contracted = false',
    3, 9, false);
```

```
 seq | path_seq | node | edge | cost | agg_cost
-----+----------+------+------+------+----------
   1 |        1 |    3 |    3 |    1 |        0
   2 |        2 |    4 |   16 |    1 |        1
   3 |        3 |    9 |   -1 |    0 |        2
(3 rows)
```

Future Directions

Some ideas (not a part of the proposal) which can be implemented for the future, include the following:

- Addition of other potential contraction algorithms to the framework.
- Implementing incremental contraction algorithms.
- Adding a functionality to pgrouting for using a contracted graph.