Graph Analytics at Scale using SparkSQL and GraphFrames

Overview

- What is Spark?
- SparkSQL & GraphFrames
- What is meant by “scale”?
- Graph Algorithms in GraphFrames
- Walkthrough: the Alternating Algorithm (Connected Components)
- Demo

What is Spark?

From the Apache Spark website:

> “Unified analytics engine for large-scale data processing”

For our purposes a few things you need to know:

- Defines an API (in several programming languages) for manipulating data on distributed computing systems
- Core data abstraction in Spark is a resilient-distributed dataset (RDD)

Spark SQL

Spark SQL sits on top of Spark core to provide a higher-level data abstraction called a DataFrame.

- Provides data users a familiar interface to query data (SQL) in addition to the Spark API
- Structure implied by DataFrames allows for additional optimizations and these optimizations are used regardless of which programming language you use

GraphFrames

Graphframes is an open-source project (not part of Spark Core) which provides DataFrame-based graphs.

- Utilizes DataFrames instead of RDDs to execute a subset of graph algorithms
- Implements a core set of graph algorithms on data at scale
- Functionality is similar to GraphX
    - “GraphX is to RDDs as GraphFrames are to DataFrames.”

What is meant by “scale”?

1. From a hardware perspective, you can improve the performance of your application by:
    - Using more powerful machines (vertical scaling)
    - Using more machines (horizontal scaling)

2. From a software perspective, the algorithms you choose to implement can execute on graphs with billions of nodes and edges

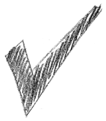
So… does GraphFrames “scale”?

1. From a hardware perspective, you can improve the performance of your application by:
    - Using more powerful machines (vertical scaling)
    - Using more machines (horizontal scaling)

2. From a software perspective, the algorithms you choose to implement can execute on graphs with billions of nodes and edges

So… does GraphFrames “scale”?

1. From a hardware perspective, you can improve the performance of your application by:
    - Using more powerful machines (vertical scaling)
    - Using more machines (horizontal scaling)



2. From a software perspective, the algorithms you choose to implement can execute on graphs with billions of nodes and edges

So… does GraphFrames "scale"?

1. From a hardware perspective, you can improve the performance of your application by:
    - Using more powerful machines (vertical scaling)
    - Using more machines (horizontal scaling)

2. From a software perspective, the algorithms you choose to implement can execute on graphs with billions of nodes and edges

GraphFrames

The core data structure in the GraphFrames library is, unsurprisingly, a GraphFrame. It's a class that is instantiated with two SparkSQL DataFrames:

- Vertices (Nodes) - A DataFrame of all the unique vertices/nodes of your graph
- Edges (Linkages) - A DataFrame of all the edges/linkages between the vertices/nodes in your graph

GraphFrames

```python
# Vertex DataFrame
v = spark.createDataFrame([
  ("a", "Alice", 34),
  ("b", "Bob", 36),
  ("c", "Charlie", 30),
  ("d", "David", 29),
  ("e", "Esther", 32),
  ("f", "Fanny", 36),
  ("g", "Gabby", 60)
], ["id", "name", "age"])
```

```python
# Edge DataFrame
e = spark.createDataFrame([
  ("a", "b", "friend"),
  ("b", "c", "follow"),
  ("c", "b", "follow"),
  ("f", "c", "follow"),
  ("e", "f", "follow"),
  ("e", "d", "friend"),
  ("d", "a", "friend"),
  ("a", "e", "friend")
], ["src", "dst", "relationship"])
# Create a GraphFrame
g = GraphFrame(v, e)
```

```python
# Create a GraphFrame
g = GraphFrame(v, e)
```

GraphFrames

```python
# Vertex DataFrame
v = spark.createDataFrame([
  ("a", "Alice", 34),
  ("b", "Bob", 36),
  ("c", "Charlie", 30),
  ("d", "David", 29),
  ("e", "Esther", 32),
  ("f", "Fanny", 36),
  ("g", "Gabby", 60)
], ["id", "name", "age"])
```

```python
# Edge DataFrame
e = spark.createDataFrame([
  ("a", "b", "friend"),
  ("b", "c", "follow"),
  ("c", "b", "follow"),
  ("f", "c", "follow"),
  ("e", "f", "follow"),
  ("e", "d", "friend"),
  ("d", "a", "friend"),
  ("a", "e", "friend")
], ["src", "dst", "relationship"])
# Create a GraphFrame
g = GraphFrame(v, e)
```

```python
# Create a GraphFrame
g = GraphFrame(v, e)
```

Graph Algorithms in GraphFrames

- Breadth-first Search
- Connected Components
    - Strongly Connected Components
- Label Propagation Algorithm
- PageRank
- Shortest Path
- Triangle Count

See the docs for more details on each of these.

Connected Components

For a given node, what other nodes are connected to it, either directly or indirectly?

Connected Components algorithms make explicit connections between nodes in a graph by assigning each node a cluster ID. Nodes with the same cluster IDs are connected

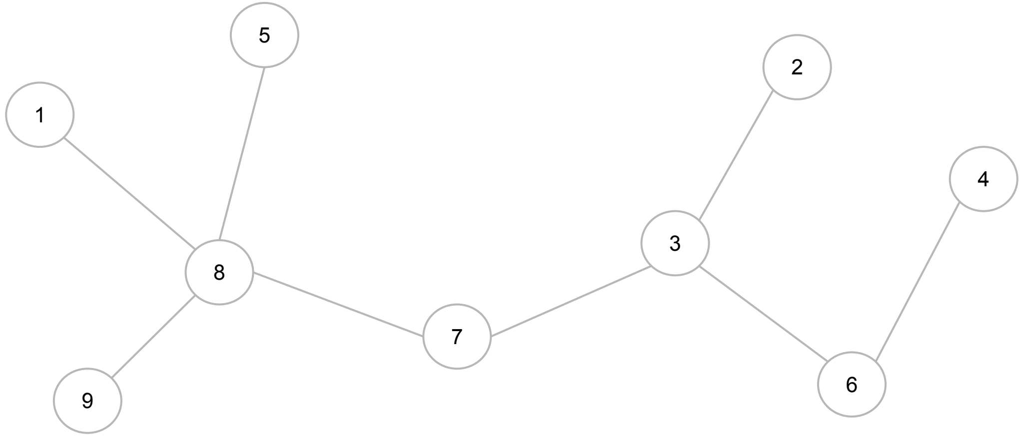
Connected Components - Motivation

Nodes	**Edges**	
ID	*From*	*To*
1	1	8
2	5	8
3	8	9
4	7	8
5	3	7
6	2	3
7	3	6
8	4	6
9		

Connected Components - Motivation

Nodes	Edges	
ID	*From*	*To*
1	1	8
2	5	8
3	8	9
4	7	8
5	3	7
6	2	3
7	3	6
8	4	6
9		

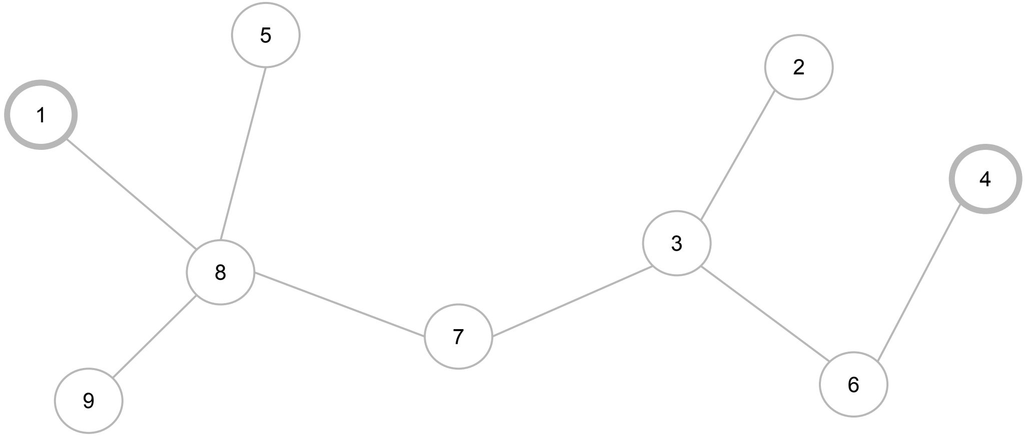
Connected Components - Motivation



Nodes	Edges	
ID	*From*	*To*
1	1	8
2	5	8
3	8	9
4	7	8
5	3	7
6	2	3
7	3	6
8	4	6
9		

Connected Components - Motivation

From the edges table, how would we know that nodes 1 and 4 are connected?



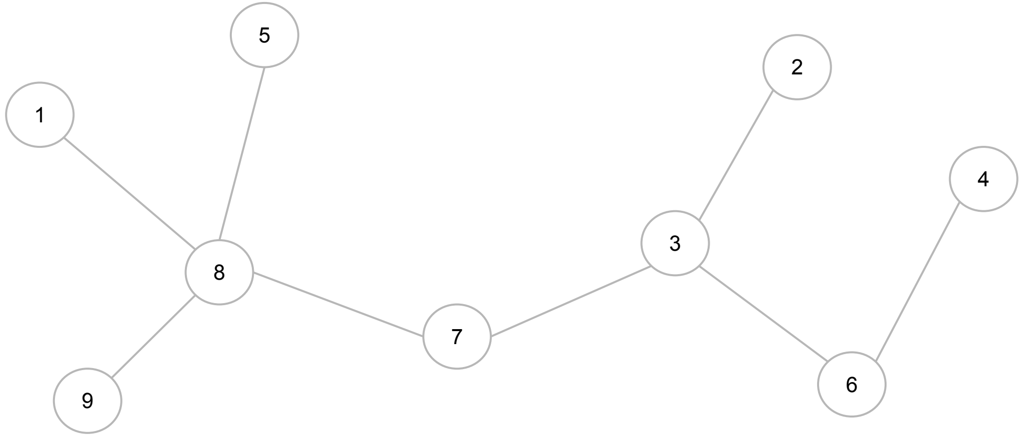
Nodes	Edges	
ID	*From*	*To*
1	1	8
2	5	8
3	8	9
4	7	8
5	3	7
6	2	3
7	3	6
8	4	6
9		

Connected Components - Motivation

We know if nodes are connected only if there is a <u>direct</u> connection in the edge table. Any <u>indirect</u> connections that may exist can be determined via a connected components algorithm.

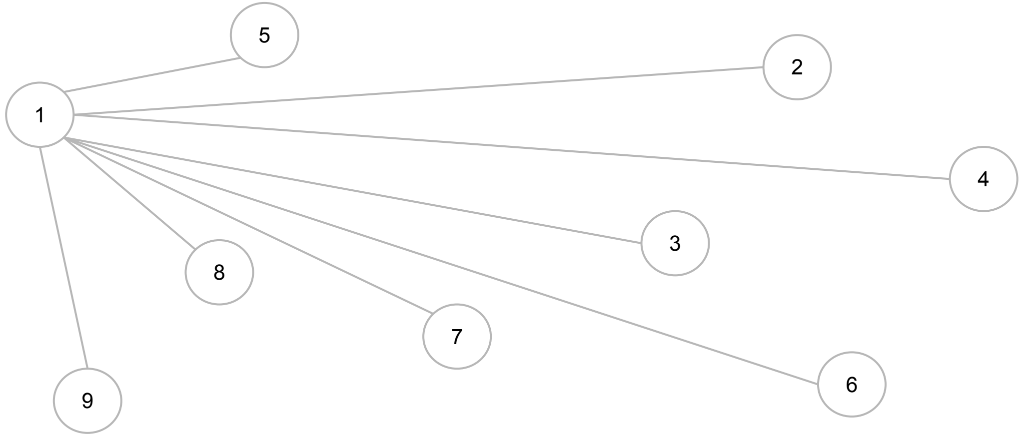
One approach is to iteratively reformulate the edge table until all nodes that are connected, either directly or indirectly, are made to be explicitly connected directly

Connected Components - Motivation



Nodes	Edges	
ID	*From*	*To*
1	1	8
2	5	8
3	8	9
4	7	8
5	3	7
6	2	3
7	3	6
8	4	6
9		

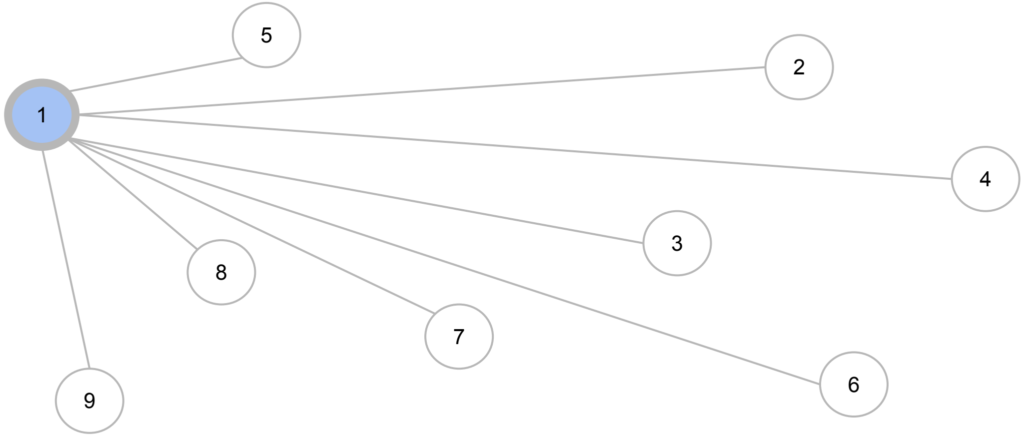
Connected Components - Motivation



Nodes	Edges	
ID	*From*	*To*
1	1	2
2	1	3
3	1	4
4	1	5
5	1	6
6	1	7
7	1	8
8	1	9
9		

Connected Components - Motivation

Cluster ID: 1



Nodes	Edges	
ID	*From*	*To*
1	1	2
2	1	3
3	1	4
4	1	5
5	1	6
6	1	7
7	1	8
8	1	9
9		

Connected Components

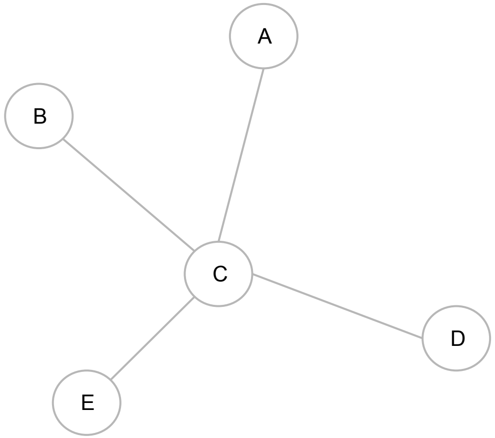
The Connected Component algorithm implemented in GraphFrames is called the Alternating Algorithm, as defined in the Google research paper from 2014:

Connected Components in MapReduce and Beyond

This algorithm works by iteratively reformulating the edge table by alternating two operations - the Large Star operation and the Small Star operation - until convergence.

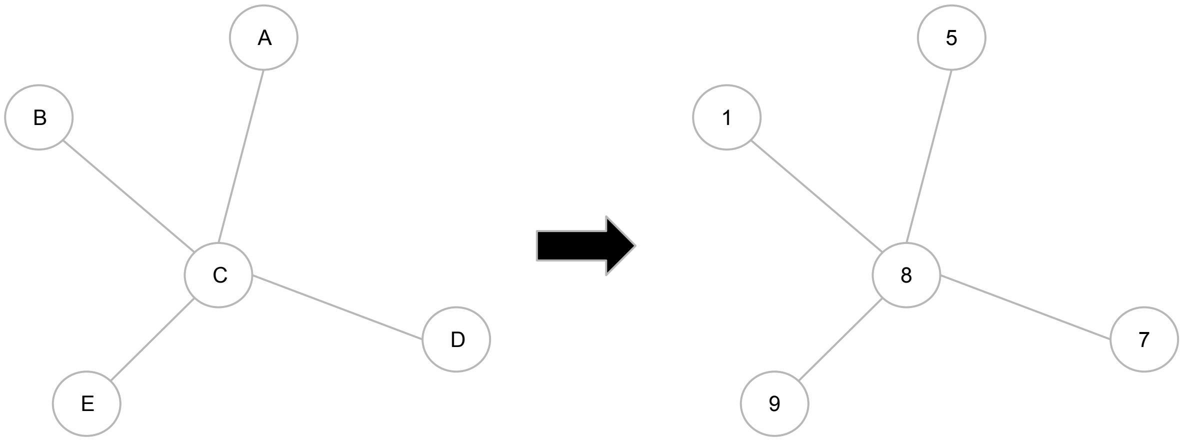
Alternating Algorithm - Initialization

Randomly assign integer IDs to all nodes in graph



Alternating Algorithm - Initialization

Randomly assign integer IDs to all nodes in graph

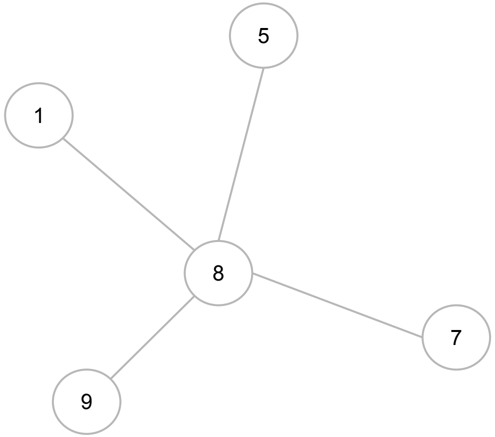


Large Star Operation

For each node in the graph, connect all strictly larger neighbors to the min neighbor (including self)

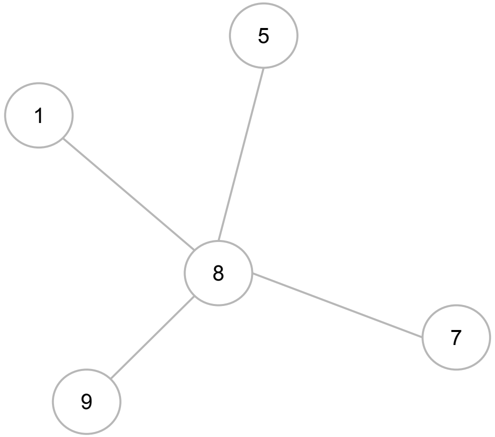
Large Star Operation

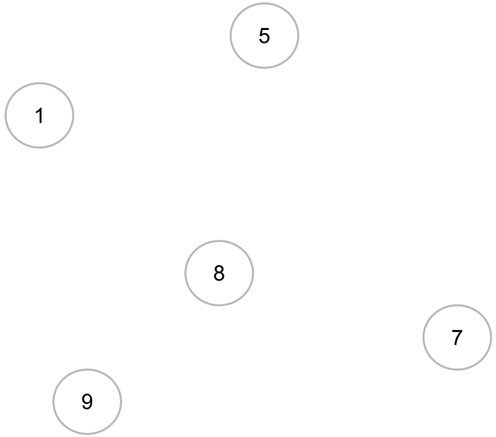
For each node in the graph, connect all strictly larger neighbors to the min neighbor (including self)



Large Star Operation

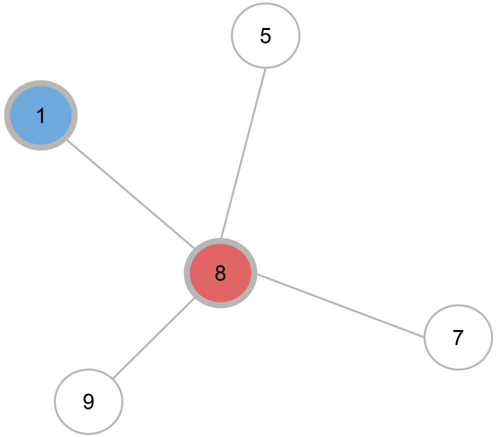
For each node in the graph, connect all strictly larger neighbors to the min neighbor (including self)

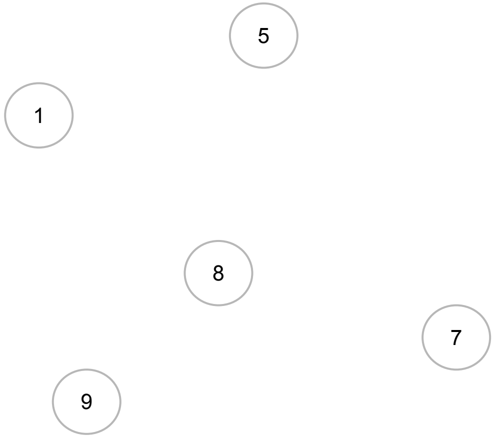




Large Star Operation

For each node in the graph, connect all strictly larger neighbors to the min neighbor (including self)



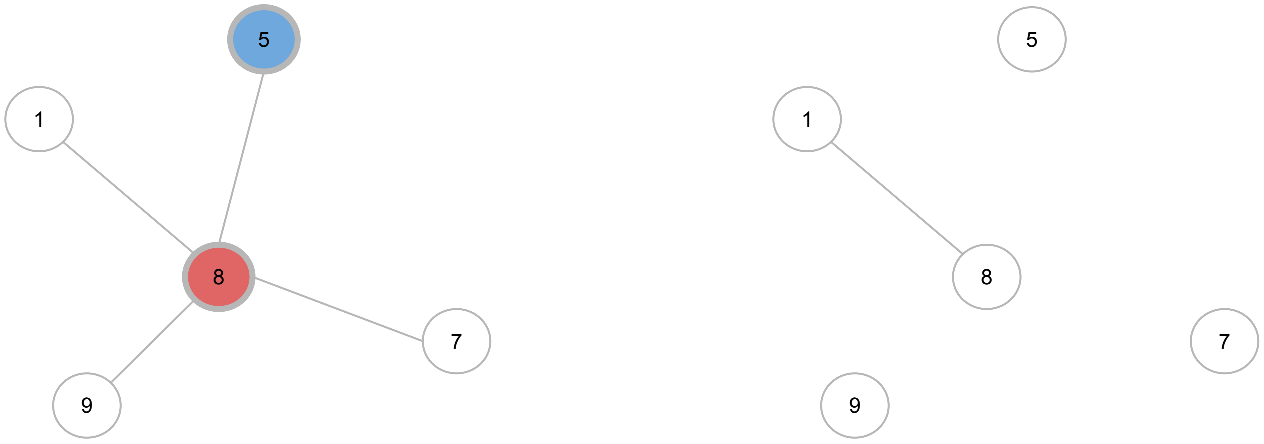


Large Star Operation

For each node in the graph, connect all strictly larger neighbors to the min neighbor (including self)

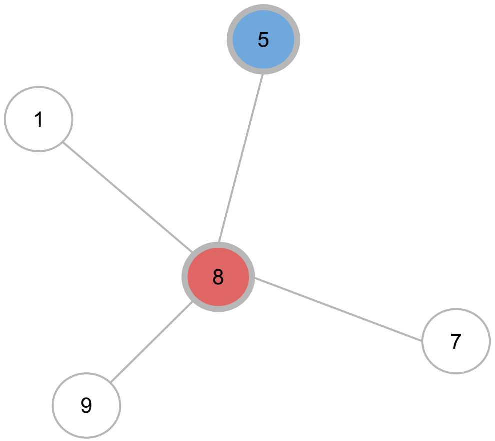
Large Star Operation

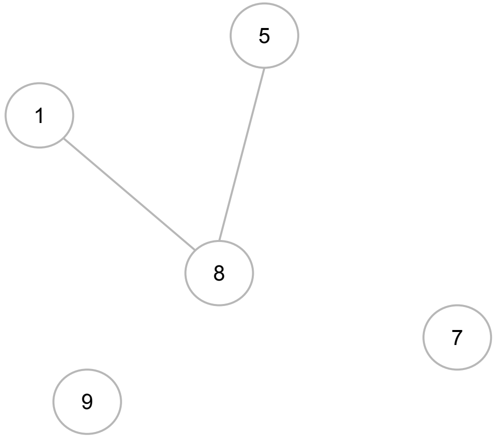
For each node in the graph, connect all strictly larger neighbors to the min neighbor (including self)



Large Star Operation

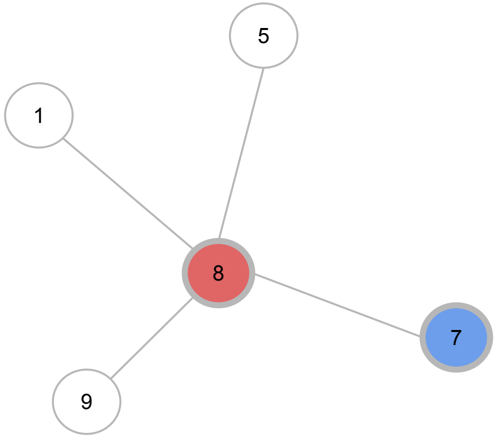
For each node in the graph, connect all strictly larger neighbors to the min neighbor (including self)

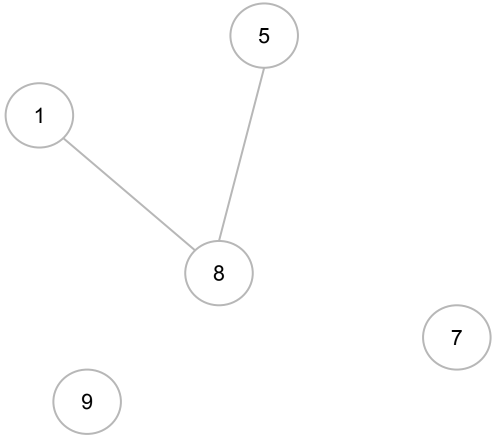




Large Star Operation

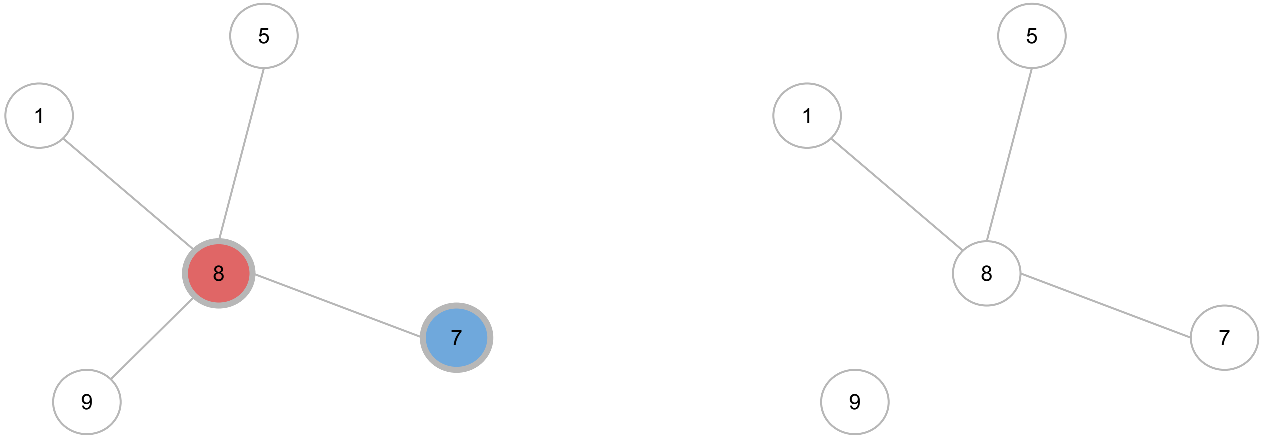
For each node in the graph, connect all strictly larger neighbors to the min neighbor (including self)





Large Star Operation

For each node in the graph, connect all strictly larger neighbors to the min neighbor (including self)

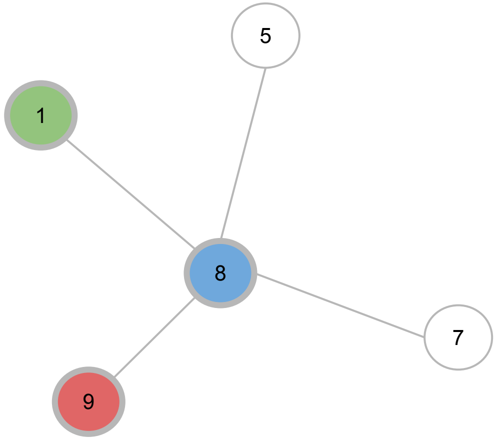


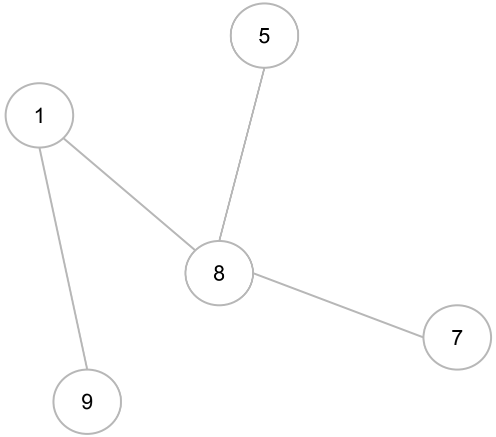
Large Star Operation

For each node in the graph, connect all strictly larger neighbors to the min neighbor (including self)

Large Star Operation

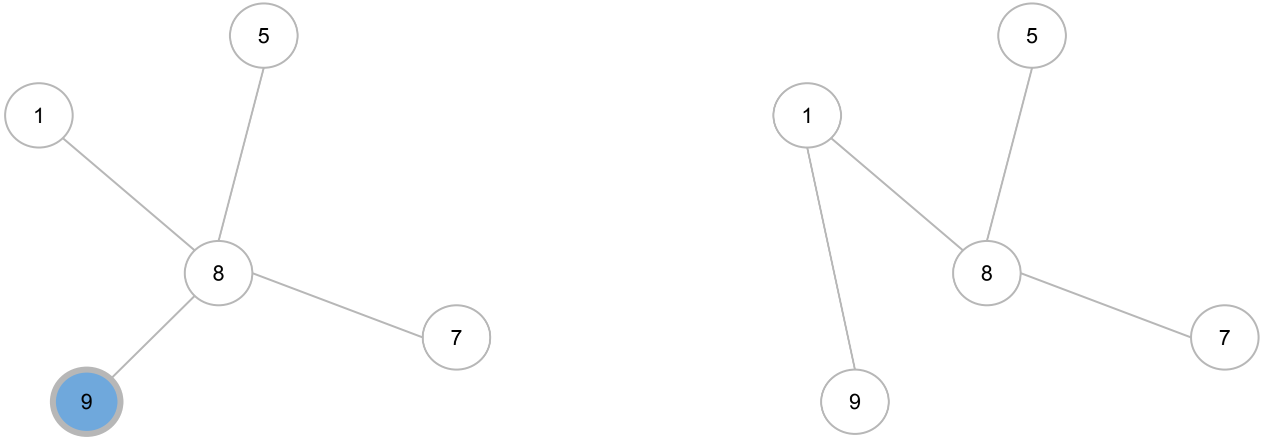
For each node in the graph, connect all strictly larger neighbors to the min neighbor (including self)





Large Star Operation

For each node in the graph, connect all strictly larger neighbors to the min neighbor (including self)



Large Star Operation

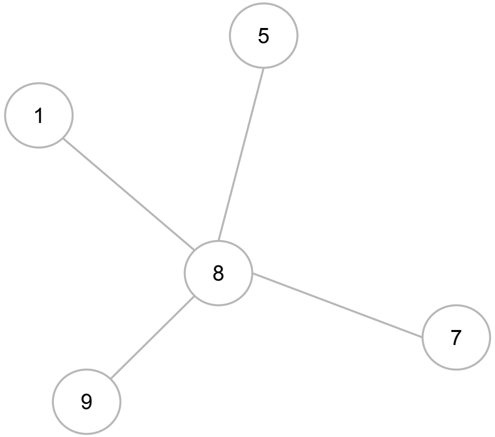
The large star operation has two theoretical guarantees:

1. Preserves connectivity of components

2. Never increases the number of edges in the graph

Small Star Operation

For each node in the graph, connect all smaller neighbors (and self) to the min neighbor

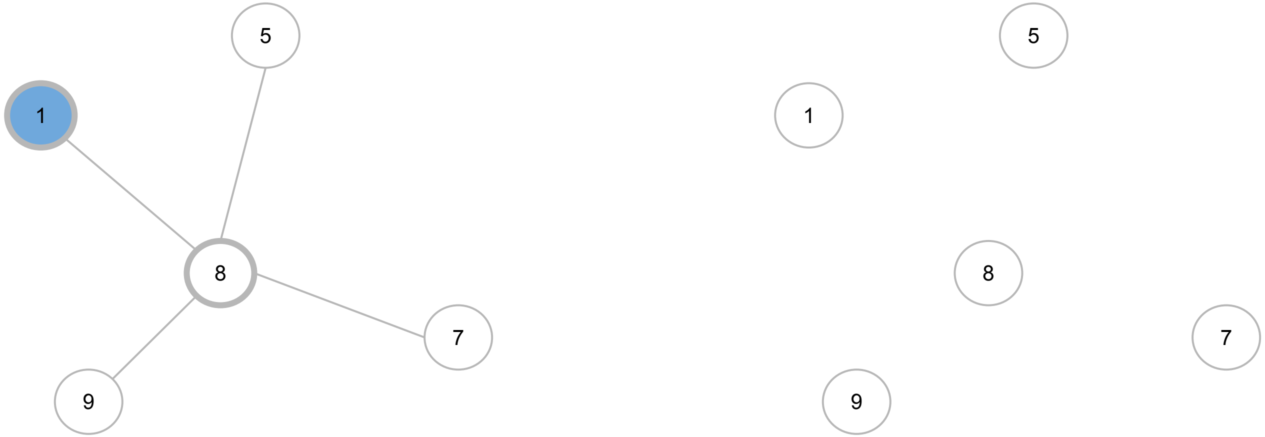


Small Star Operation

For each node in the graph, connect all smaller neighbors (and self) to the min neighbor

Small Star Operation

For each node in the graph, connect all smaller neighbors (and self) to the min neighbor

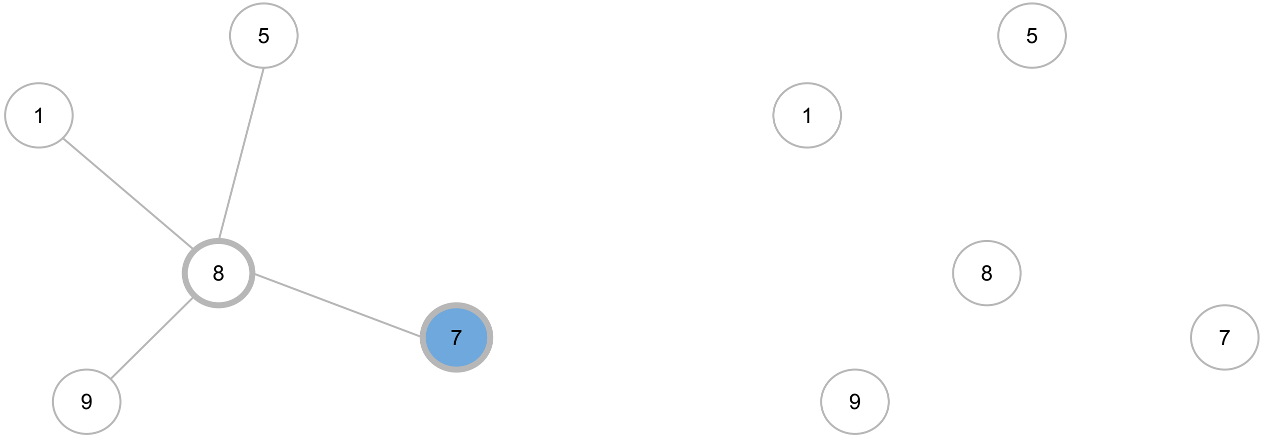


Small Star Operation

For each node in the graph, connect all smaller neighbors (and self) to the min neighbor

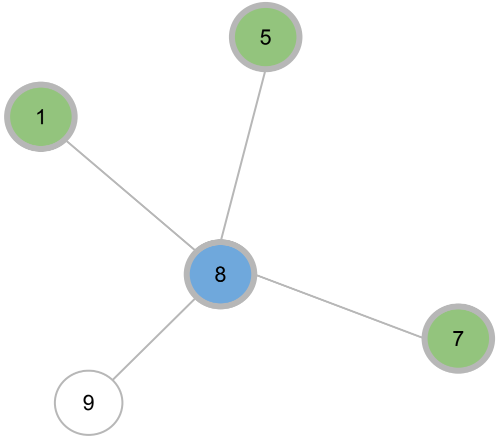
Small Star Operation

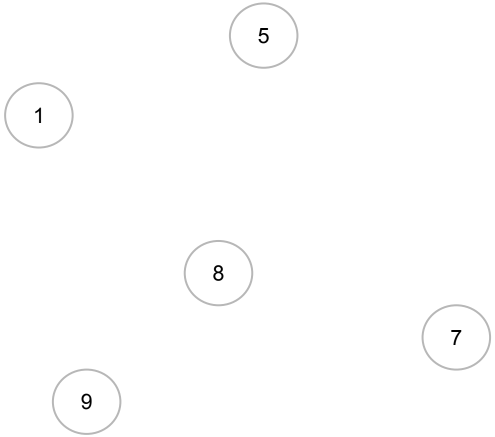
For each node in the graph, connect all smaller neighbors (and self) to the min neighbor



Small Star Operation

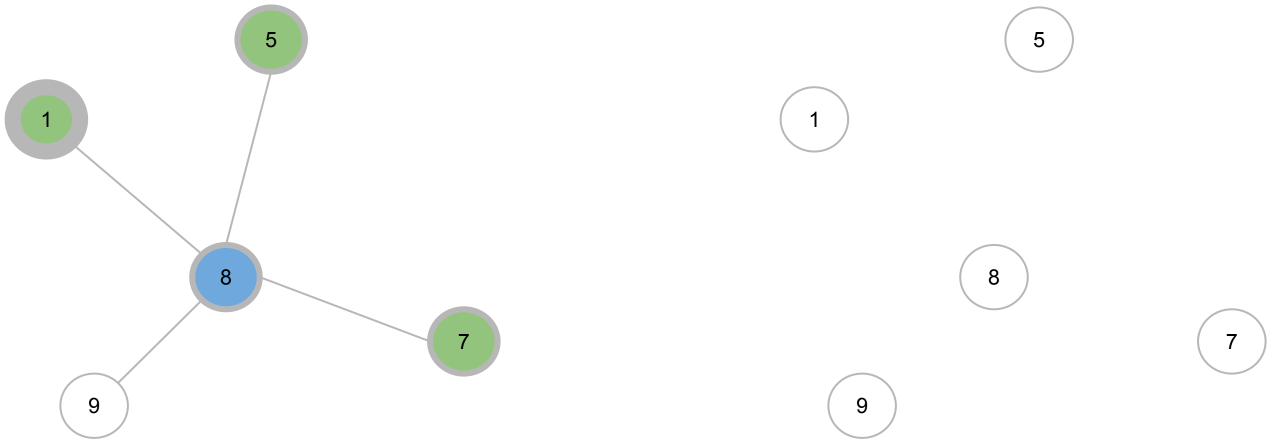
For each node in the graph, connect all smaller neighbors (and self) to the min neighbor





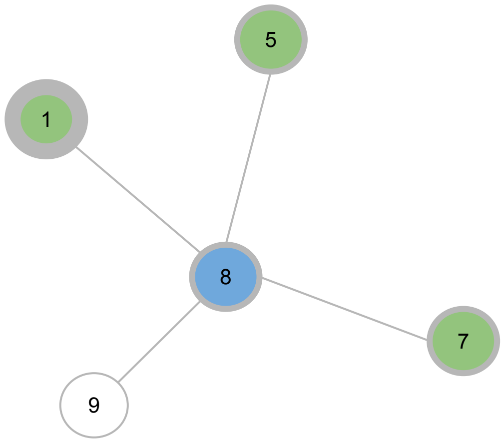
Small Star Operation

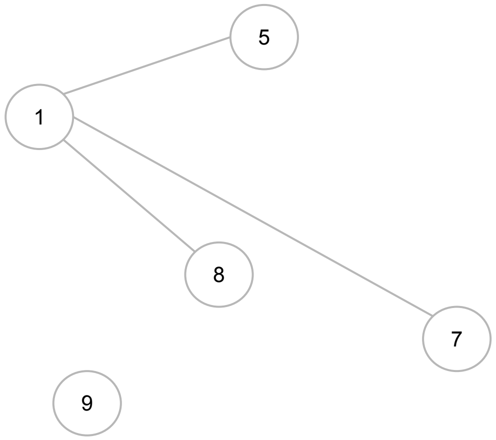
For each node in the graph, connect all smaller neighbors (and self) to the min neighbor



Small Star Operation

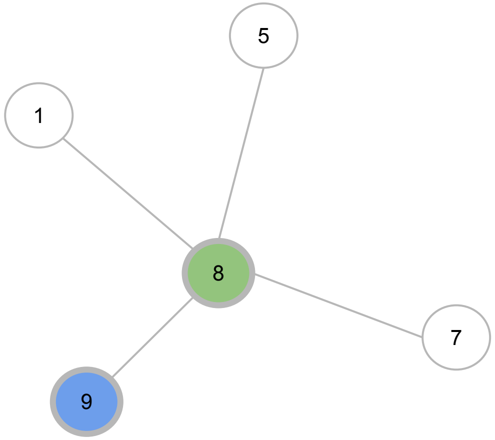
For each node in the graph, connect all smaller neighbors (and self) to the min neighbor

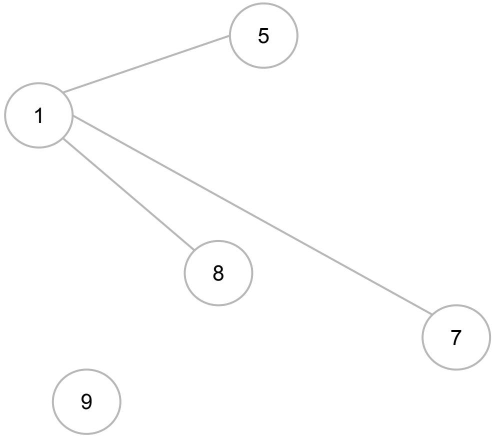




Small Star Operation

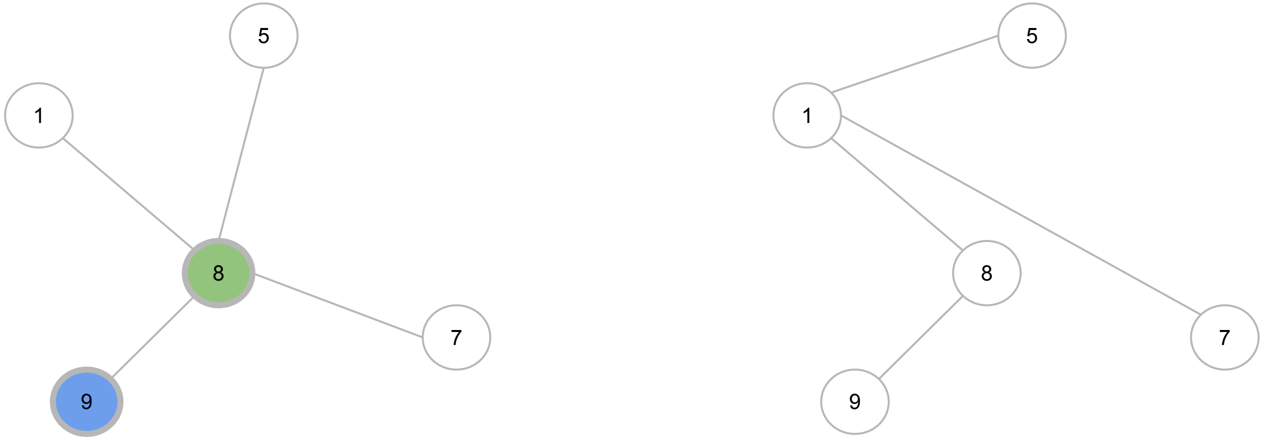
For each node in the graph, connect all smaller neighbors (and self) to the min neighbor





Small Star Operation

For each node in the graph, connect all smaller neighbors (and self) to the min neighbor



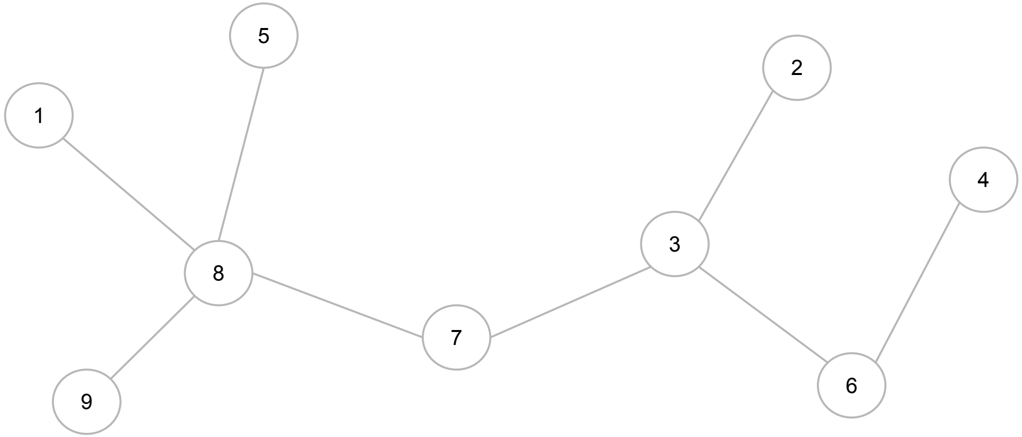
Small Star Operation

The small star operation has the same theoretical guarantees as the large star operation:

1. Preserves connectivity of components

2. Never increases the number of edges in the graph

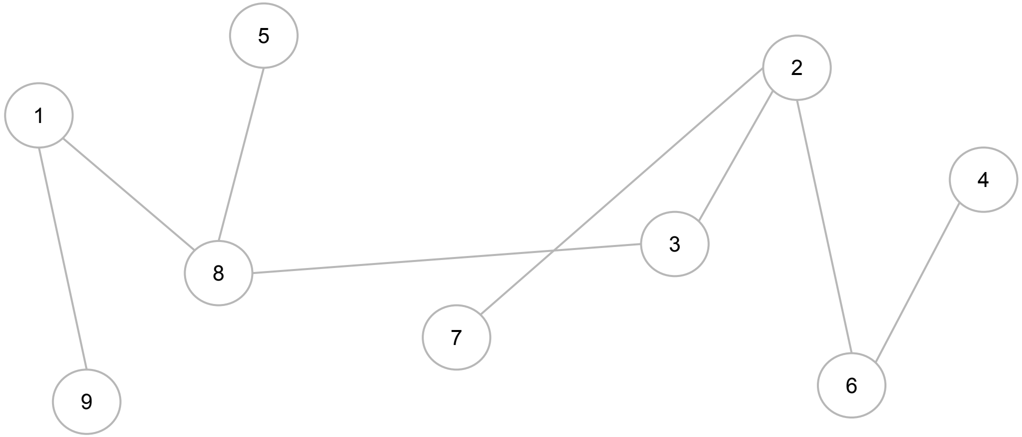
Alternating Algorithm - Example



Nodes	Edges	
ID	*From*	*To*
1	1	8
2	5	8
3	8	9
4	7	8
5	3	7
6	2	3
7	3	6
8	4	6
9		

Alternating Algorithm - Example

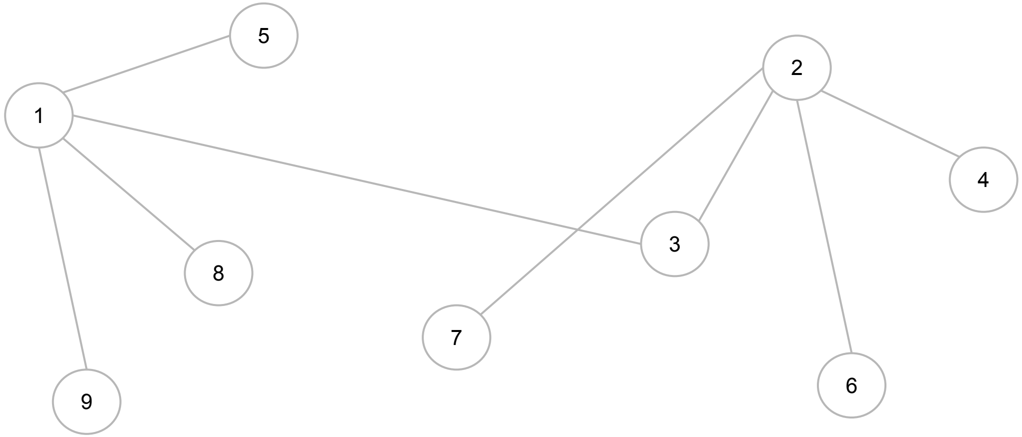
Step 1 - Large Star (1)



Nodes	Edges	
ID	*From*	*To*
1	1	9
2	1	8
3	5	8
4	3	8
5	2	7
6	2	3
7	2	6
8	4	6
9		

Alternating Algorithm - Example

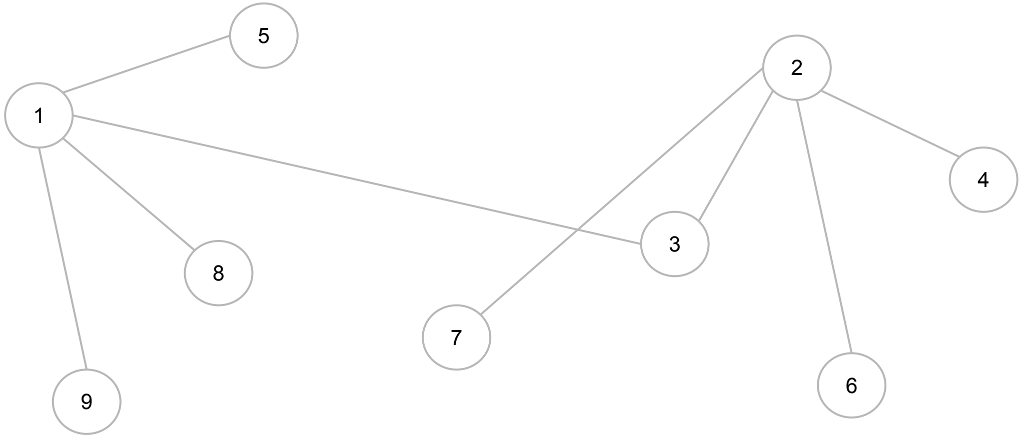
Step 2 - Small Star (1)



Nodes	Edges	
ID	*From*	*To*
1	1	9
2	1	8
3	1	5
4	1	3
5	2	7
6	2	3
7	2	6
8	2	4
9		

Alternating Algorithm - Example

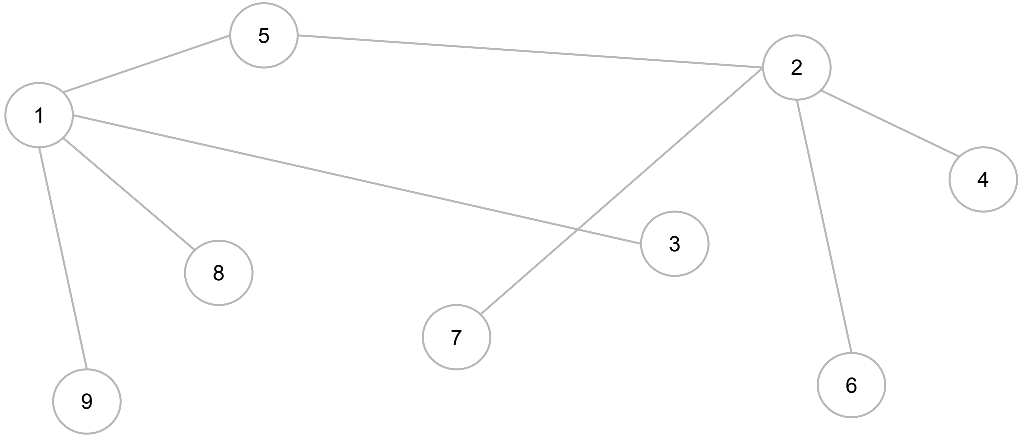
Step 3 - Large Star (2)



Nodes	Edges	
ID	*From*	*To*
1	1	9
2	1	8
3	1	5
4	1	3
5	2	7
6	2	3
7	2	6
8	2	4
9		

Alternating Algorithm - Example

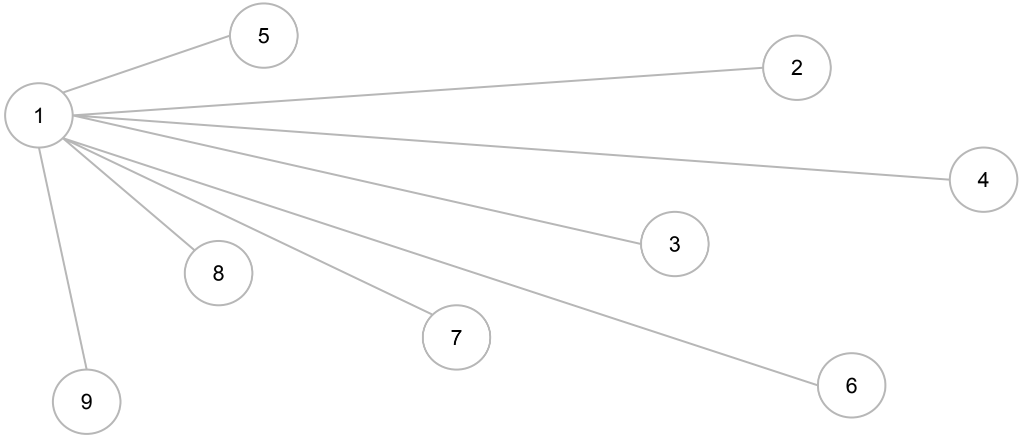
Step 4 - Small Star (2)



Nodes	Edges	
ID	*From*	*To*
1	1	9
2	1	8
3	1	5
4	1	3
5	2	7
6	2	5
7	2	6
8	2	4
9		

Alternating Algorithm - Example

Step 5 - Large Star (2)



Nodes	Edges	
ID	*From*	*To*
1	1	9
2	1	8
3	1	5
4	1	3
5	1	7
6	1	5
7	1	6
8	1	4
9		

GraphFrames - Large Star and Small Star

```
while (!converged) {
  // large-star step
  // compute min neighbors (including self-min)
  val minNbrs1 = minNbrs(ee) // src >= min_nbr
    .persist(intermediateStorageLevel)
  // connect all strictly larger neighbors to the min neighbor (including self)
  ee = skewedJoin(ee, minNbrs1, broadcastThreshold, logPrefix)
    .select(col(DST).as(SRC), col(MIN_NBR).as(DST)) // src > dst
    .distinct()
    .persist(intermediateStorageLevel)

  // small-star step
  // compute min neighbors (excluding self-min)
  val minNbrs2 = ee.groupBy(col(SRC)).agg(min(col(DST)).as(MIN_NBR), count("*").as(CNT)) // src > min_nbr
    .persist(intermediateStorageLevel)
  // connect all smaller neighbors to the min neighbor
  ee = skewedJoin(ee, minNbrs2, broadcastThreshold, logPrefix)
    .select(col(MIN_NBR).as(SRC), col(DST)) // src <= dst
    .filter(col(SRC) =!= col(DST)) // src < dst
  // connect self to the min neighbor
  ee = ee.union(minNbrs2.select(col(MIN_NBR).as(SRC), col(SRC).as(DST))) // src < dst
    .distinct()
```

GraphFrames - Convergence

```scala
    // test convergence

    // Taking the sum in DecimalType to preserve precision.
    // We use 20 digits for long values and Spark SQL will add 10 digits for the sum.
    // It should be able to handle 200 billion edges without overflow.
    val (currSum, cnt) = ee.select(sum(col(SRC).cast(DecimalType(20, 0))), count("*")).rdd
      .map { r =>
        (r.getAs[BigDecimal](0), r.getLong(1))
      }.first()
    if (cnt != 0L && currSum == null) {
      throw new ArithmeticException(
        s"""
           |The total sum of edge src IDs is used to determine convergence during iterations.
           |However, the total sum at iteration $iteration exceeded 30 digits (1e30),
           |which should happen only if the graph contains more than 200 billion edges.
           |If not, please file a bug report at https://github.com/graphframes/graphframes/issues.
          """.stripMargin)
    }
    logInfo(s"$logPrefix Sum of assigned components in iteration $iteration: $currSum.")
    if (currSum == prevSum) {
      // This also covers the case when cnt = 0 and currSum is null, which means no edges.
      converged = true
    } else {
      prevSum = currSum
    }

    iteration += 1
  }
```

Custom Graph Algorithms

GraphFrames provides primitives for developing graph algorithms

- aggregateMessages API
- Pregel API

Example: Belief propagation

Demo