ΟΙΚΟΝΟΜΙΚΟ ΠΑΝΕΠΙΣΤΗΜΙΟ ΑΘΗΝΩΝ

ATHENS UNIVERSITY OF ECONOMICS AND BUSINESS

Big Data Systems for Graphs

Yannis Kotidis

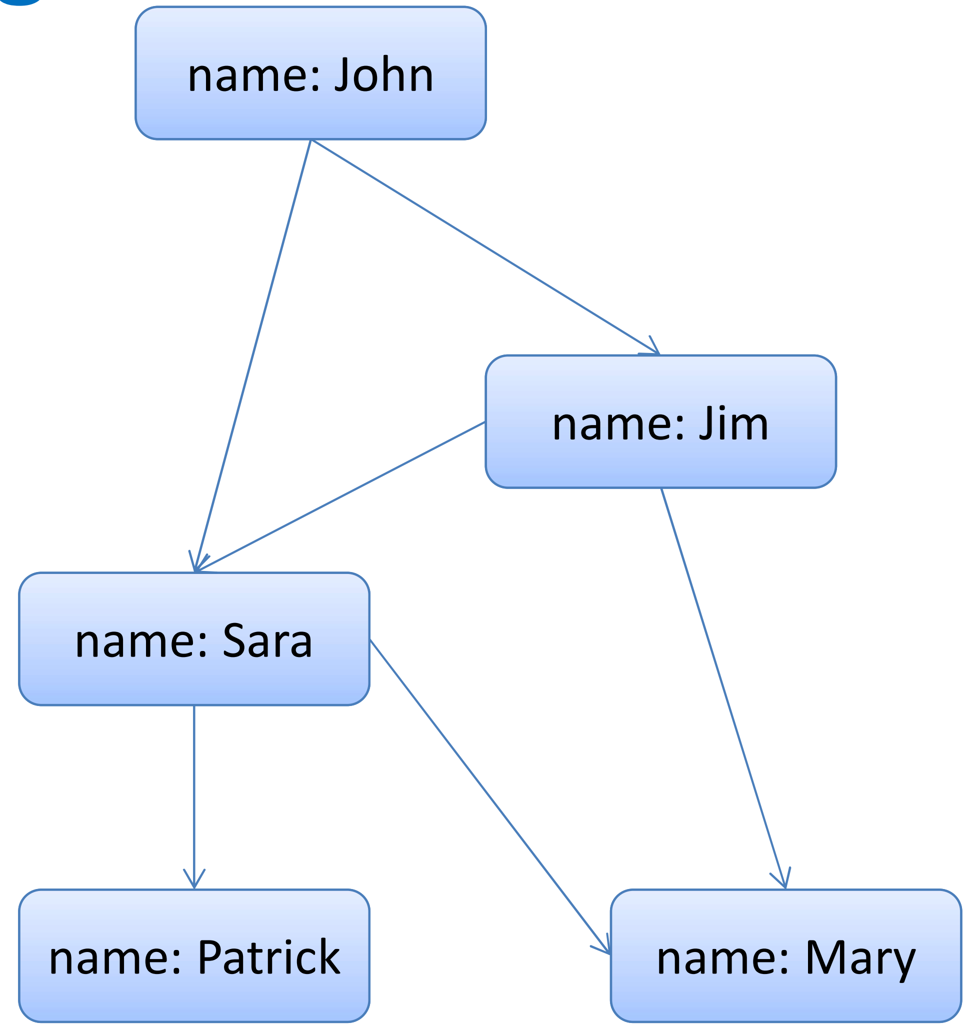
http://pages.cs.aueb.gr/~kotidis/

Apache Spark

- There are multiple ways to process graph data with Apache Spark
  - GraphX: based on RDDs
  - GraphFrames: based on DataFrames
  - Pregel API

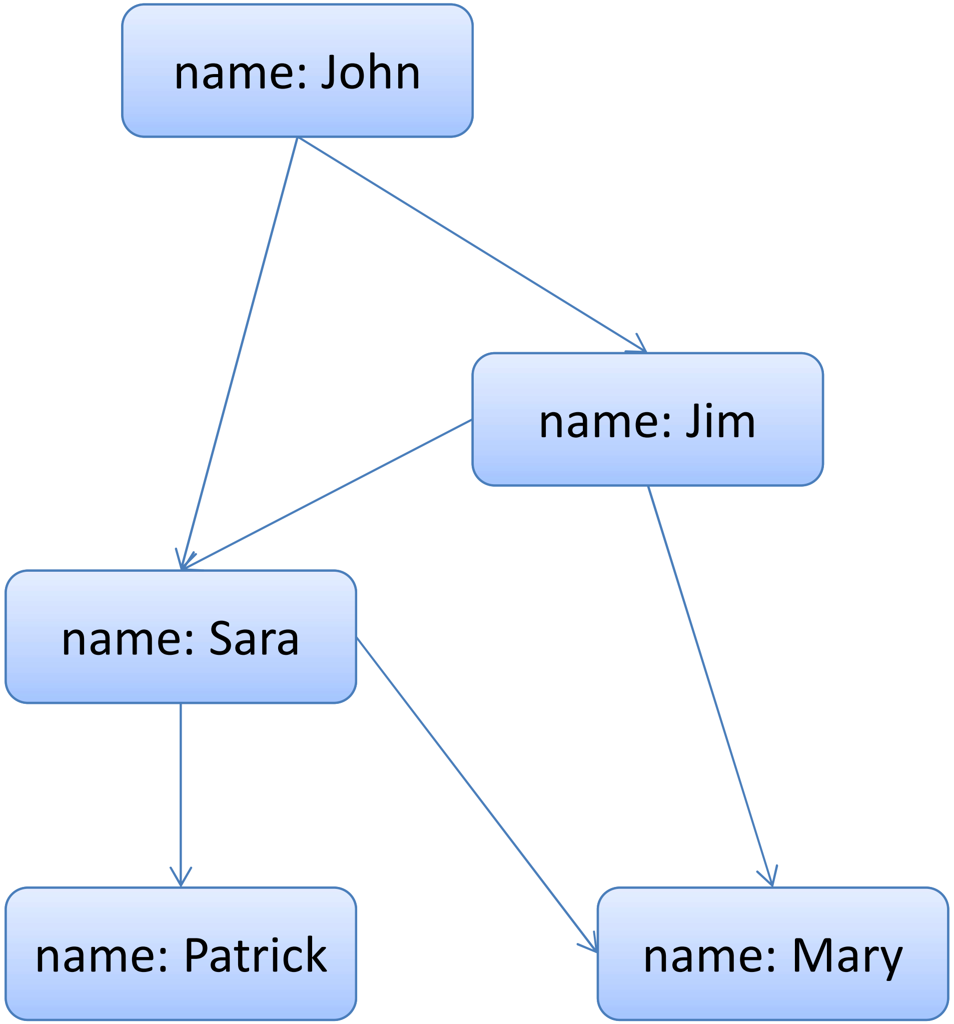
Friend suggestions example: Define nodes using a DataFrame

```
val v =
    spark.sqlContext.create
    DataFrame(List(
  ("john", "John", 29),
  ("sara", "Sara", 22),
  ("jim", "Jim", 42),
  ("patrick", "Patrick",19),
  ("mary", "Mary", 31)
)).toDF("id", "name",
    "age")
```



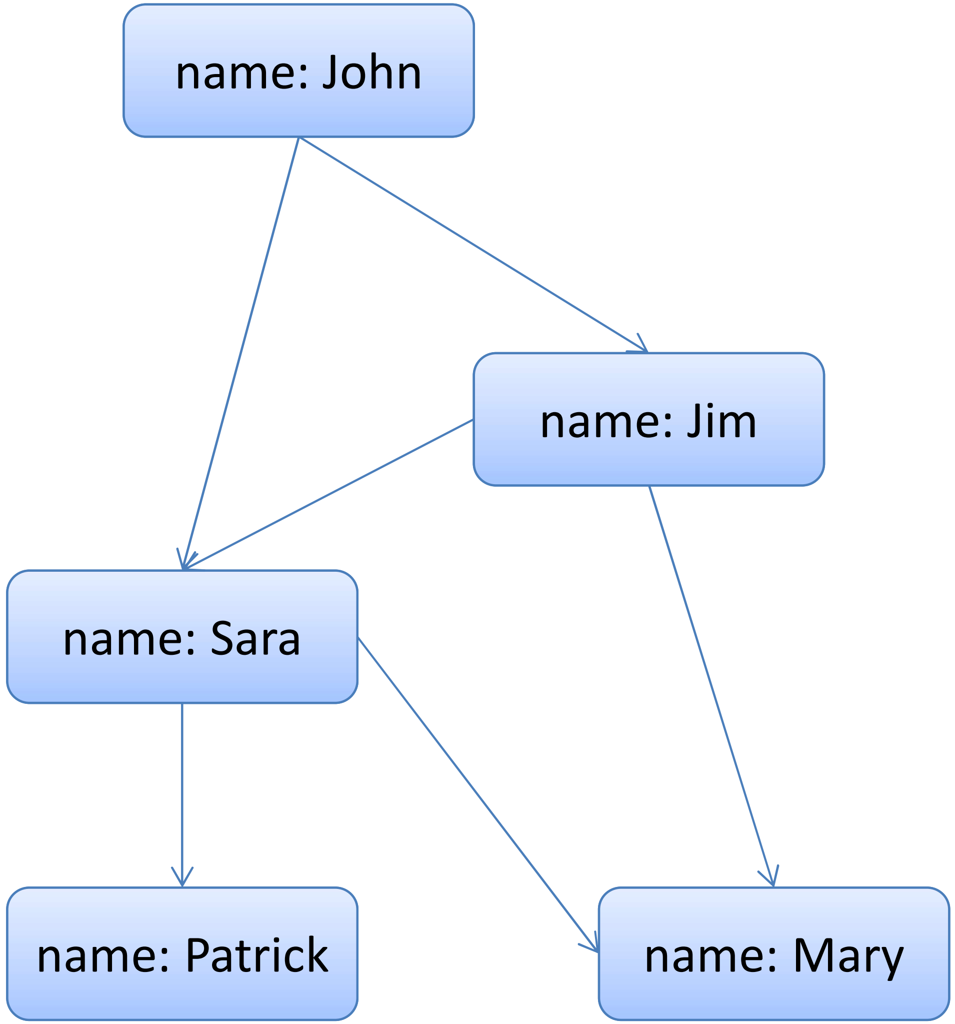
Now Define Edges

```
val e =
    spark.sqlContext.createData
    Frame(List(
  ("john", "sara", "knows"),
  ("john", "jim", "knows"),
  ("jim", "sara", "knows"),
  ("jim","mary","knows"),
  ("sara", "patrick", "knows"),
  ("sara", "mary", "knows")
)).toDF("src", "dst",
    "relationship")
```



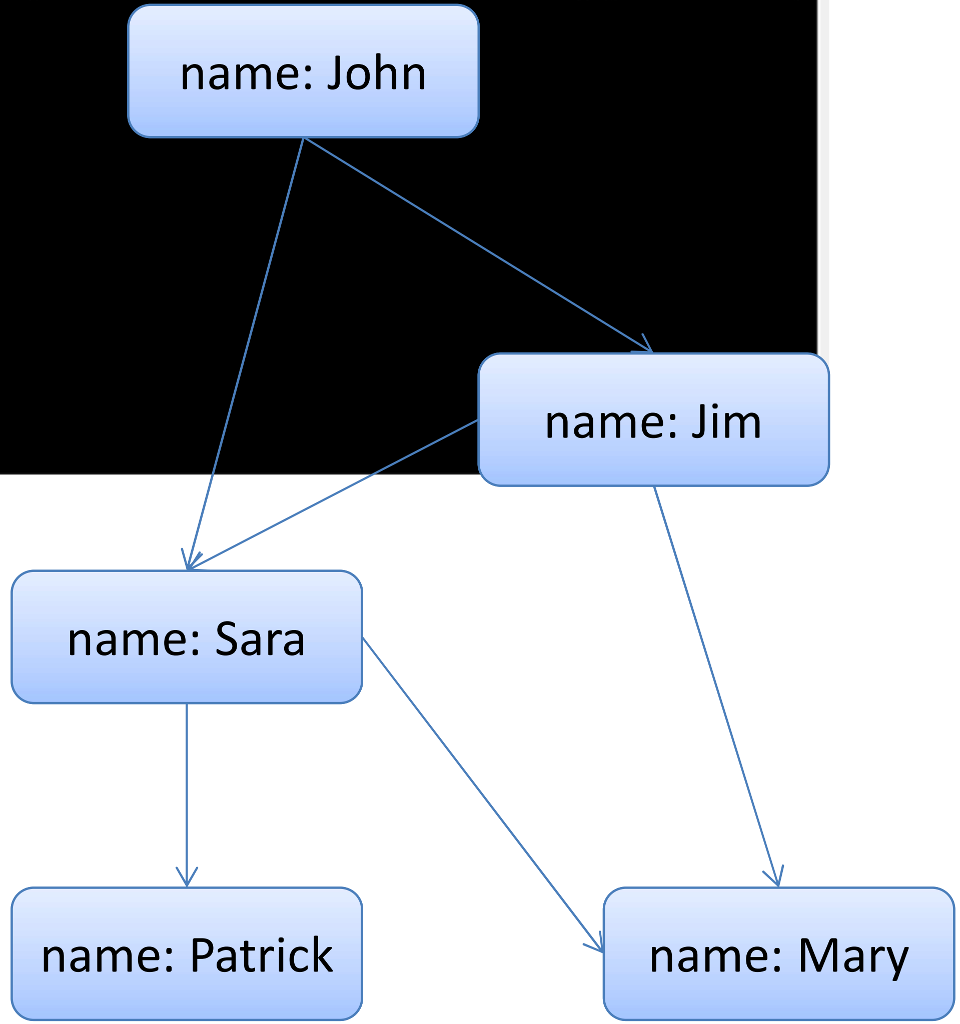
Create GraphFrame, run Motif

```
val g = GraphFrame(v, e)


g.find(
"(x)-[]->(f); (f)-[]->(fof);
   !(x)-[]->(fof)").
select("x","fof").groupBy("x
    ","fof").count.orderBy("c
    ount").show()
```



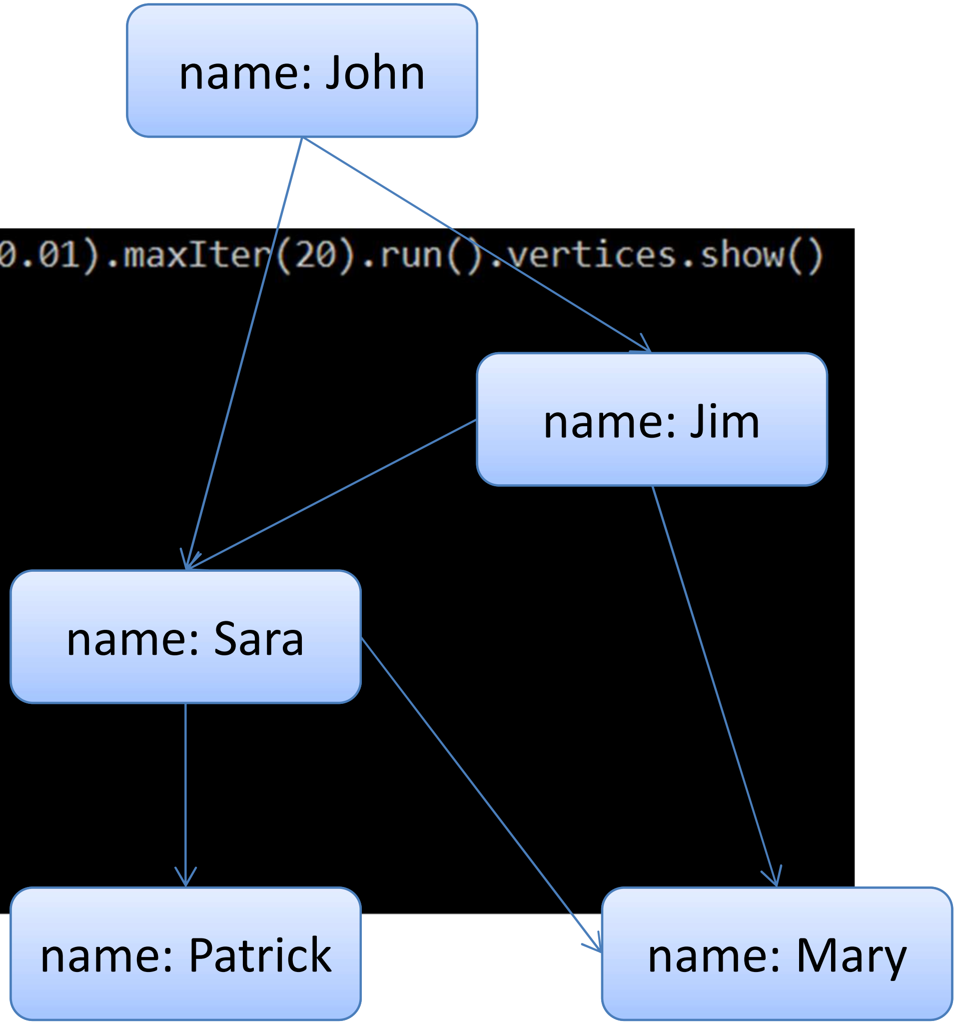
Result

```
scala> g.find("(x)-[]->(f); (f)-[]->(fof); !(x)-[]->(fof)").select("x","fof").groupBy("x","fof")
.count.orderBy("count").show()
+---------------+--------------------+-----+
|              x|                 fof|count|
+---------------+--------------------+-----+
|  [jim, Jim, 42]|[patrick, Patrick...|    1|
|[john, John, 29]|[patrick, Patrick...|    1|
|[john, John, 29]|     [mary, Mary, 31]|    2|
+---------------+--------------------+-----+

scala>
```



PageRank Example

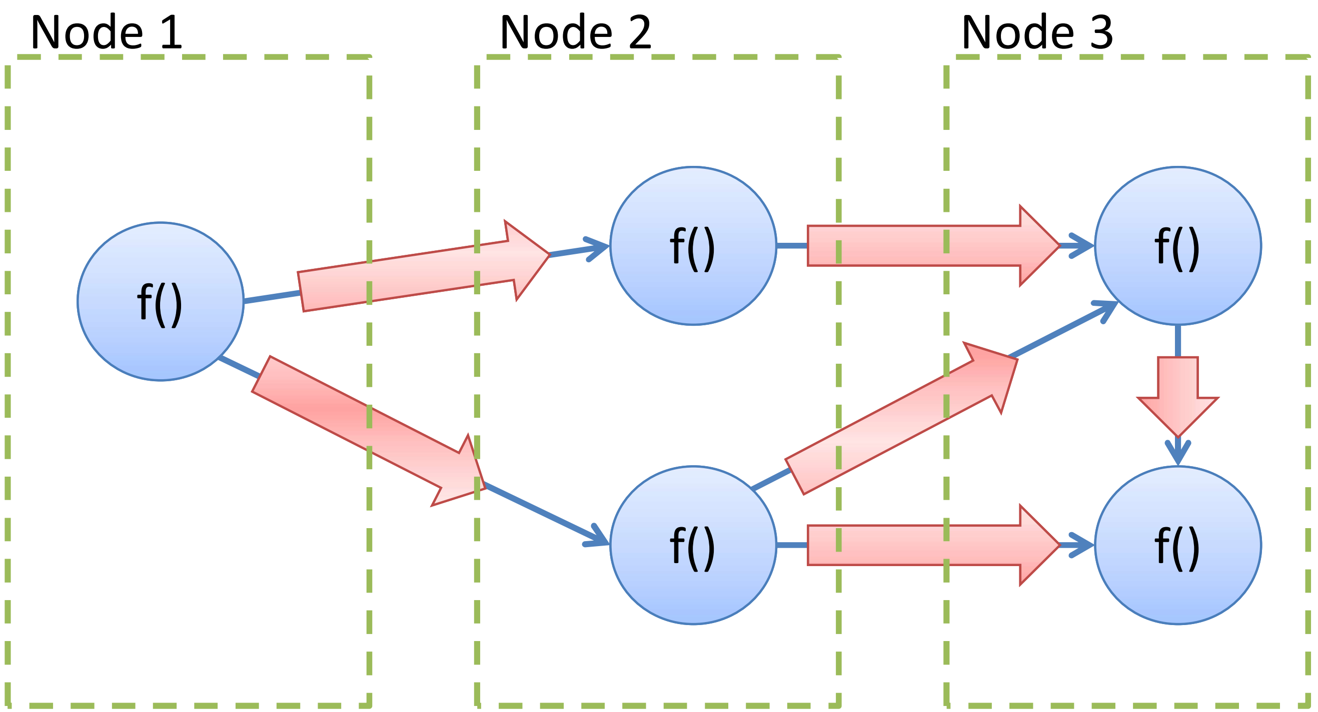
```
scala> val results = g.pageRank.resetProbability(0.01).maxIter(20).run().vertices.show()
+-------+-------+---+------------------+
|     id|   name|age|          pagerank|
+-------+-------+---+------------------+
|   mary|   Mary| 31|1.4698147724378927|
|   john|   John| 29|0.5163835727128357|
|   sara|   Sara| 22|1.1541301946025058|
|    jim|    Jim| 42|0.7719934412056895|
|patrick|Patrick| 19|1.0876780190410762|
+-------+-------+---+------------------+

results: Unit = ()

scala> 
```



Vertex-centric programming

- Distributed systems mainly deal with graph computations like shortest paths, pageRank that can be parallelized
- Key ideas
  - Implement processing logic on graph nodes (aka vertex-centric programming)
    - have all graph nodes perform the required computations in parallel
  - Sync results (message exchange phase)
  - Repeat until computation converges

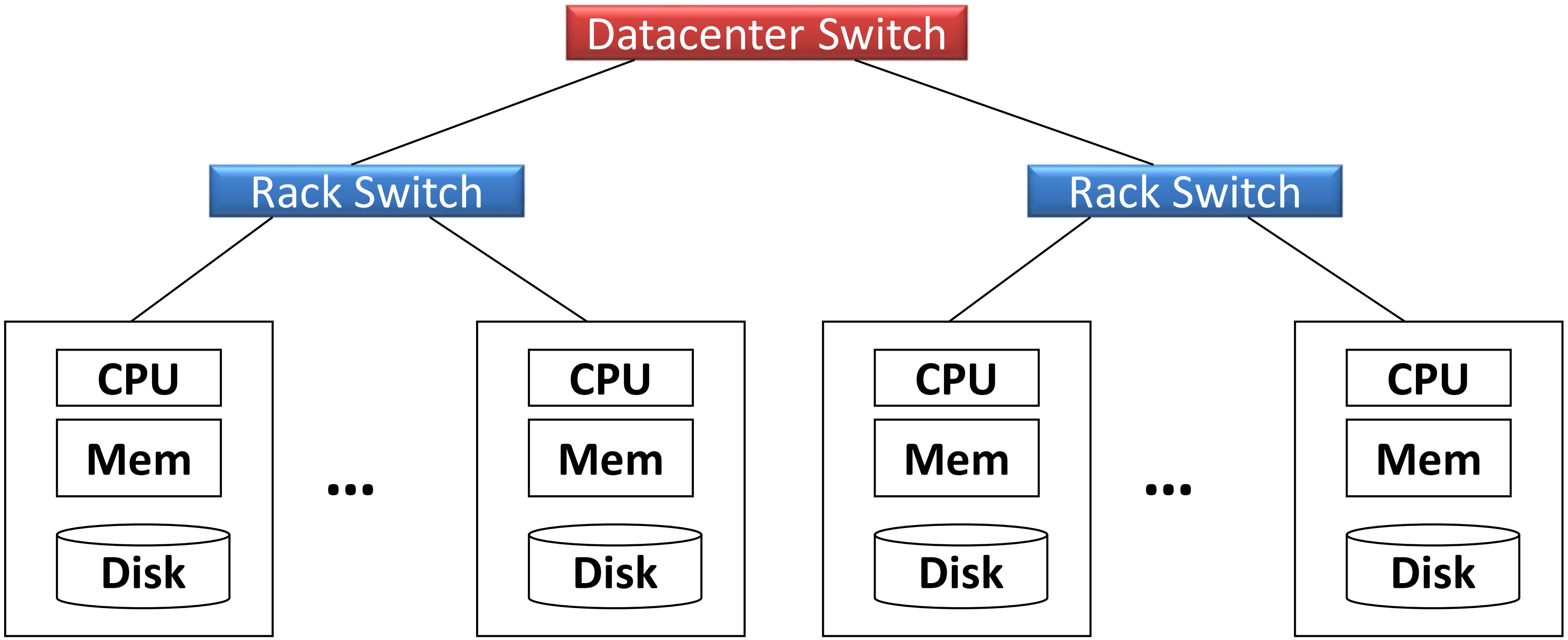
Computational Paradigm: supper steps + synchronization



- Supper step: run user-defined code f()
- Synchronization: message exchange

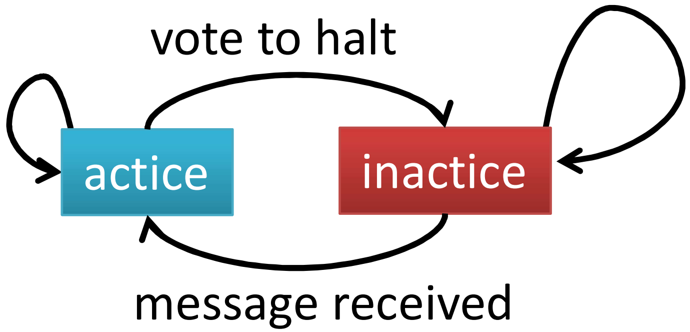
Pregel

- Pregel is a framework developed by Google.
  - System was never release to the public but has been copied once paper was out
- It was designed for the Google cluster architecture.
  - Each cluster consists of thousands of commodity PCs organized into racks with high intra-rack bandwidth
  - Clusters are interconnected but distributed geographically

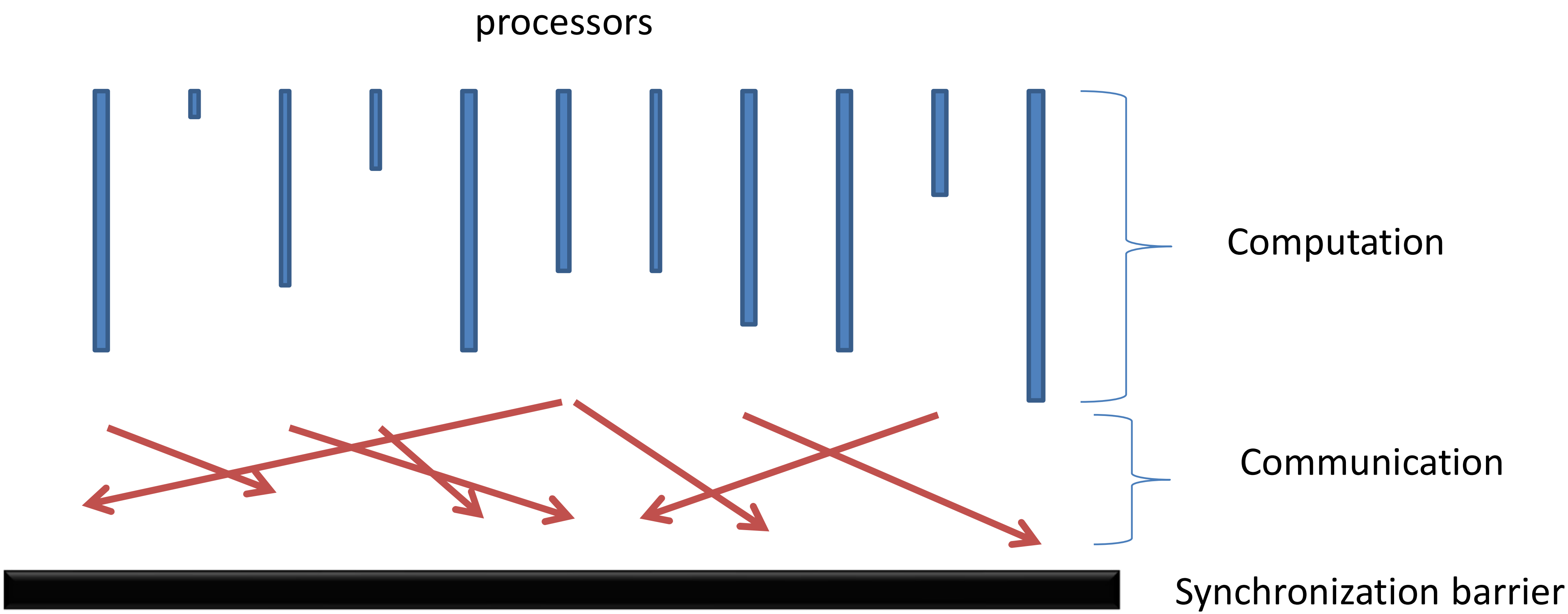


Computational Model

- All vertices compute in parallel during a superstep
  - Process messages sent in the previous superstep
  - Execute the same user-defined compute() function
  - Optionally a vertex
    - Modifies its value or that of its outgoing edges
    - Sends messages to other vertices (to be received in the next superstep)
    - Changes the topology of the graph
  - Votes to halt if it has no further work to do
- Pregel program terminates when
  - All vertices are simultaneously inactive
  - There are no messages in transit

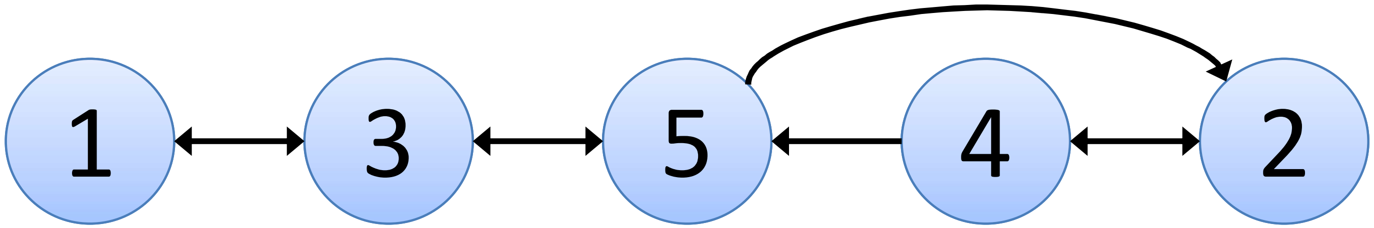


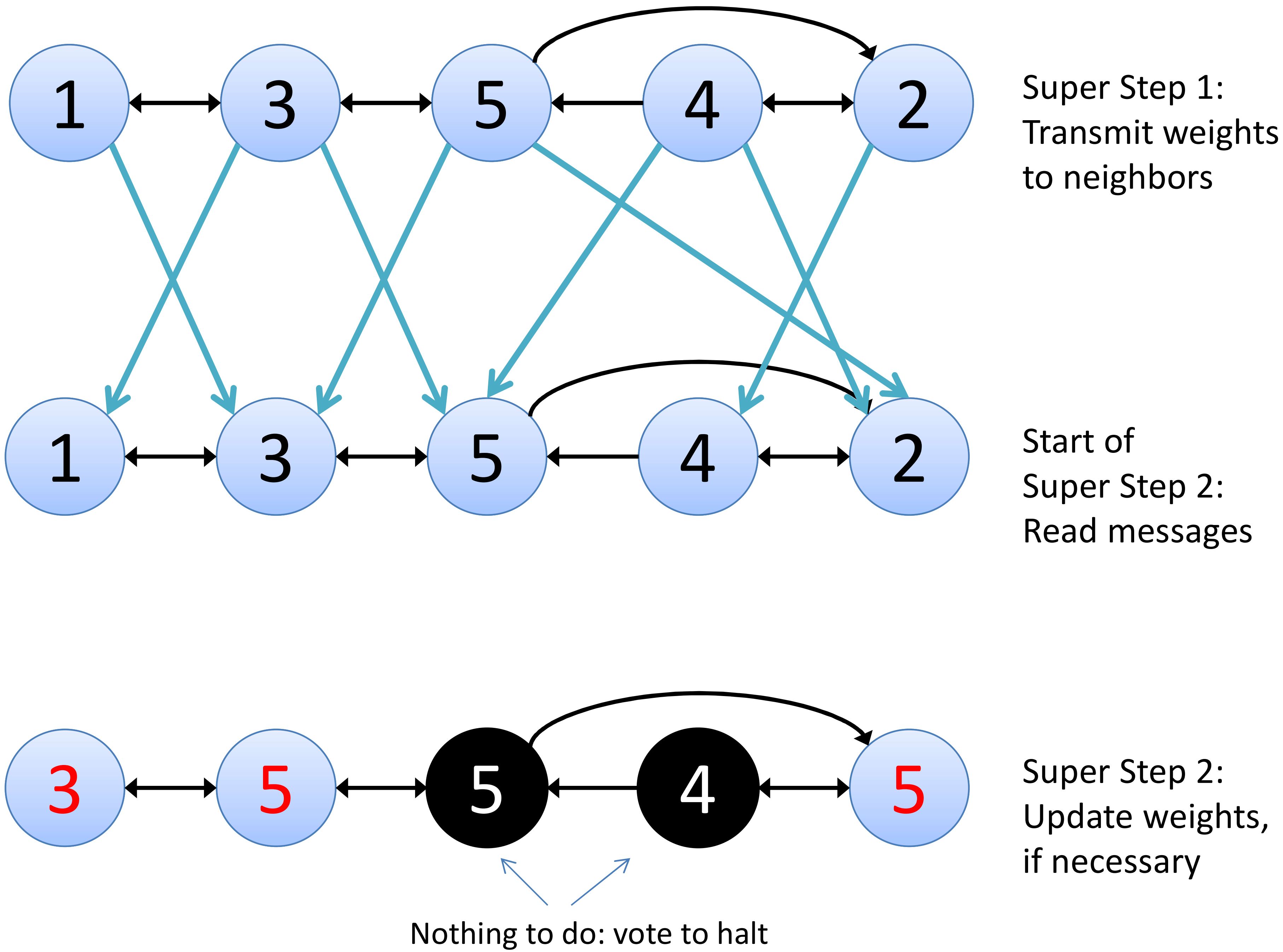
Bulk Synchronous Parallel Computing (Leslie Gabriel Valiant)

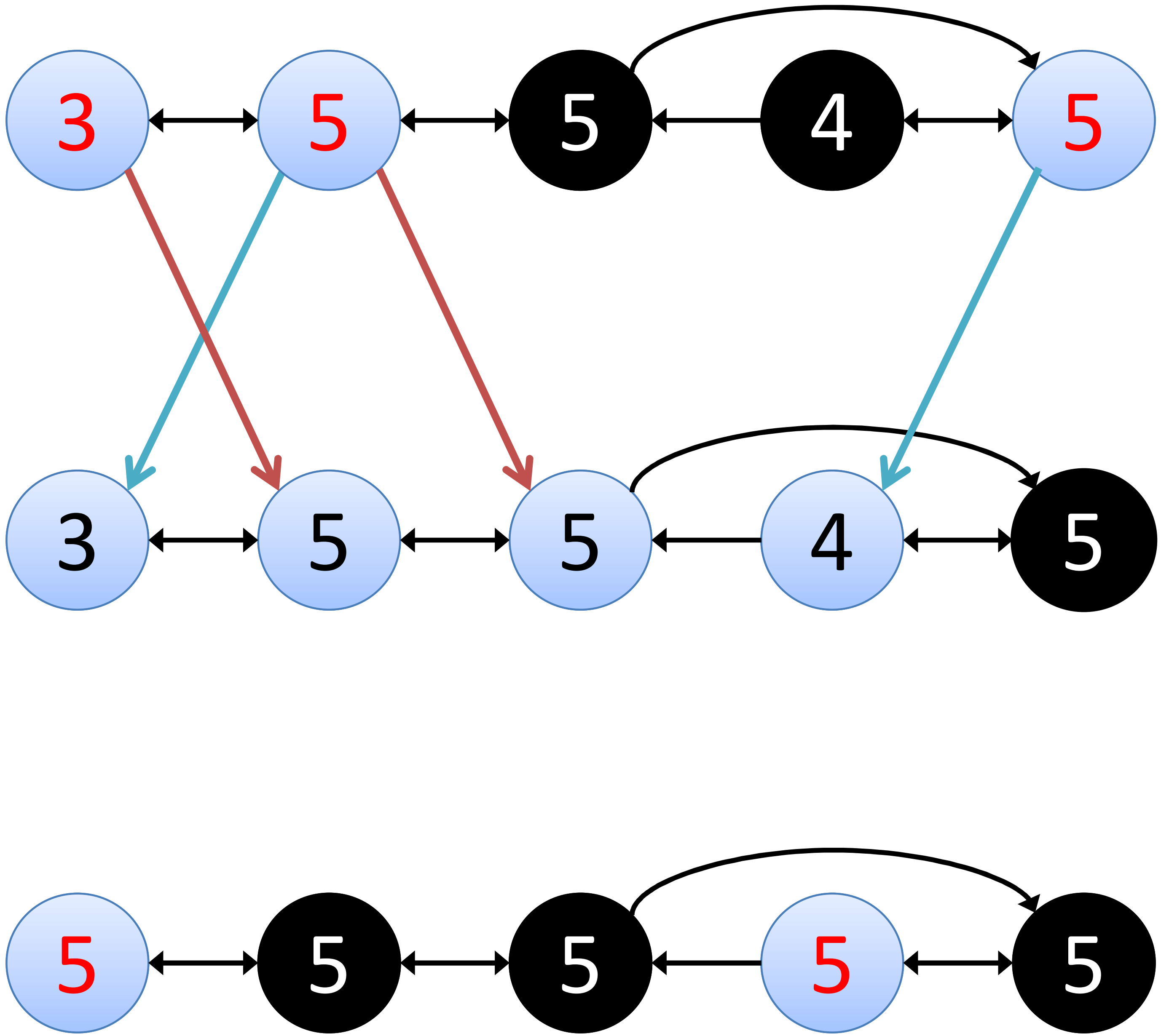


Toy problem

- Find the maximum value in a strongly connected graph component
  - Strongly connected: there is a directed path between any two vertices u, v



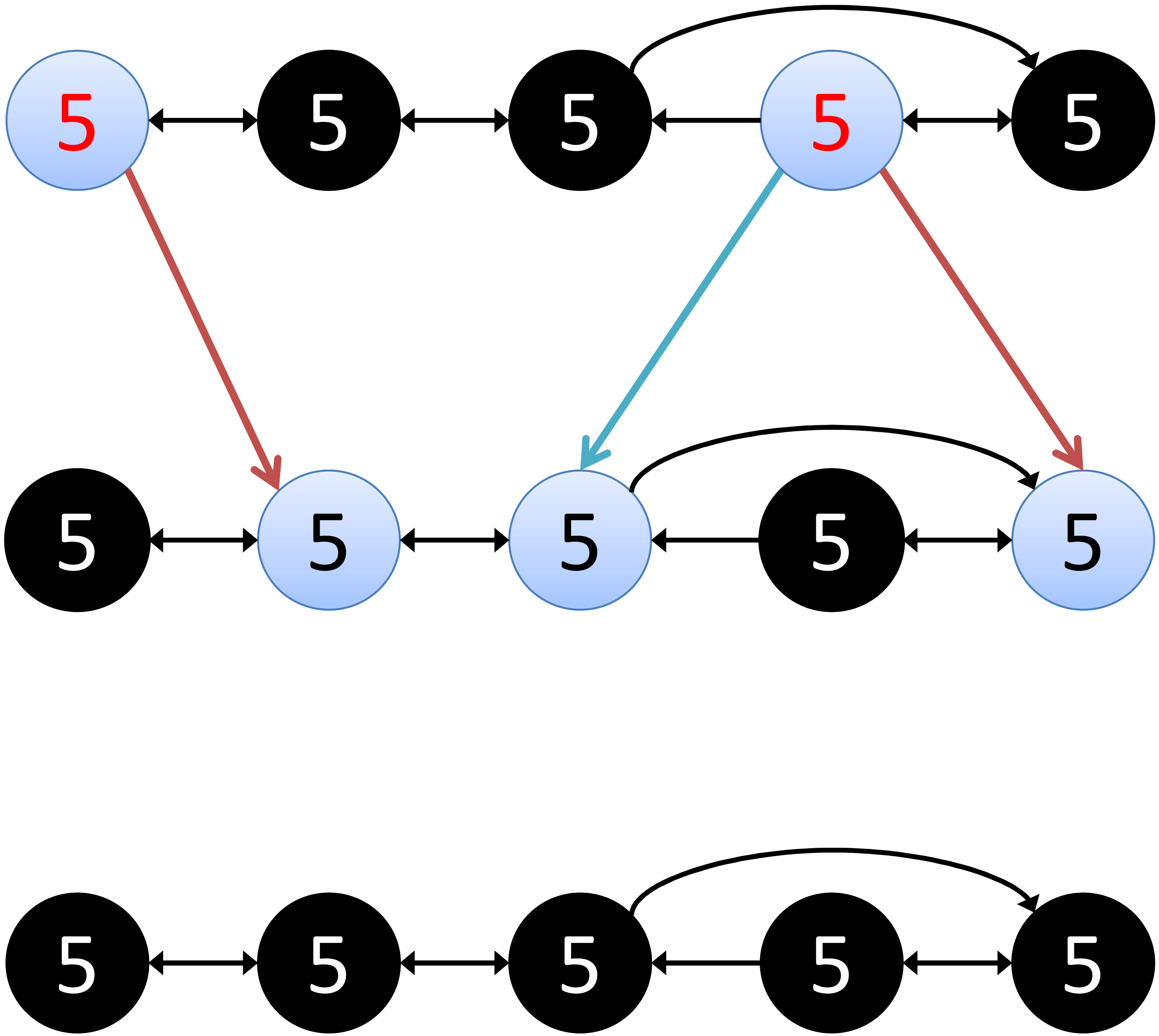

Super Step 1:
Transmit weights
to neighbors
Start of
Super Step 2:
Read messages
Super Step 2:
Update weights,
if necessary
Nothing to do: vote to halt



Super Step 2:
Transmit new weights,
if necessary

Super Step 3:
Read messages

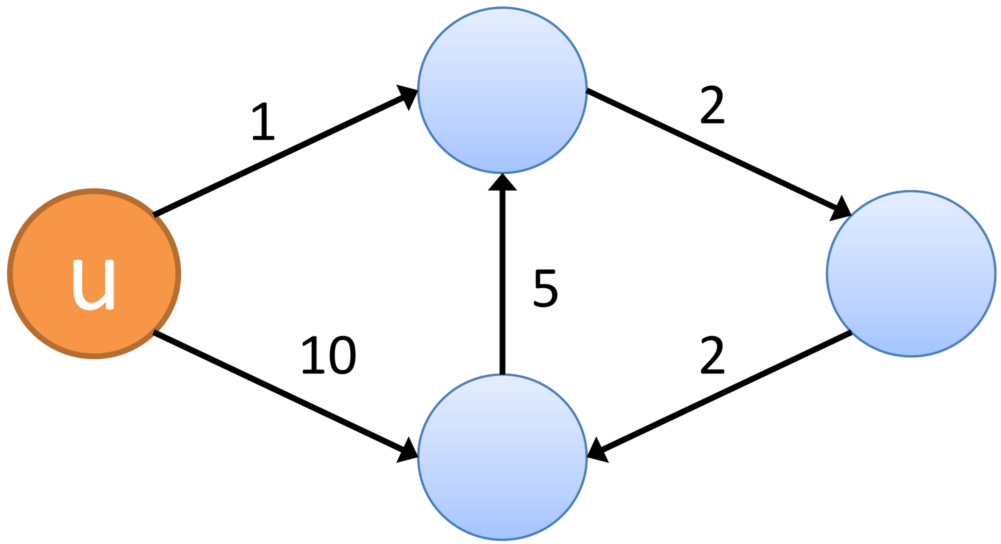
Super Step 3:
Update weights



5
5
5
5
5
5
5
5
5
5
5
5
5
5
5

Single Source Shortest Path

- Find shortest path from a source node u to all nodes
- Solution
  - Single CPU machine: Dijkstra's algorithm

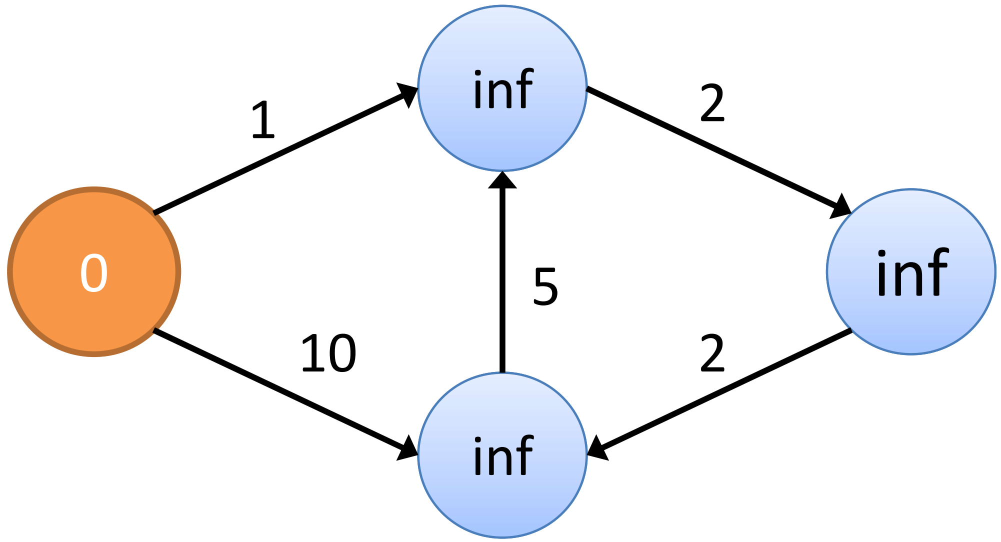


Dijkstra’s algorithm Overview

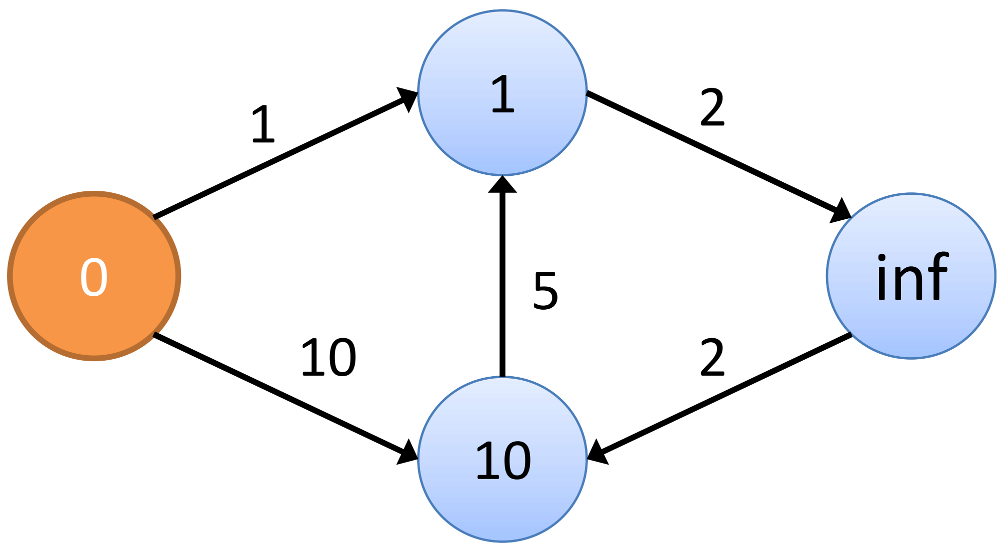
- Maintain distances of nodes from source (initially infinite, except source) in a priority queue
- At each step
  - Remove from queue node with minimum distance
  - Update shortest paths of adjacent nodes

Example: initialize queue

Q={0,inf,inf,inf}

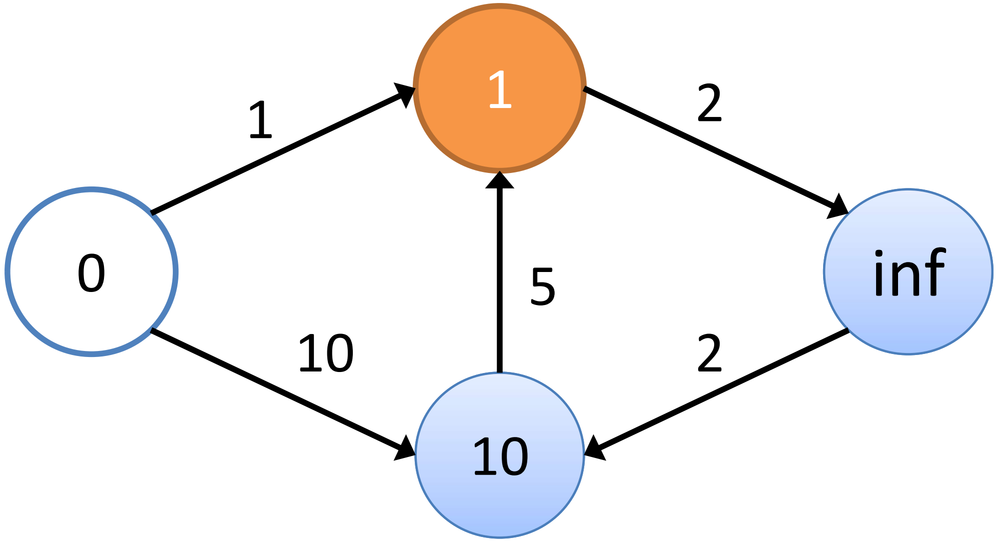


Update distances of adjacent nodes

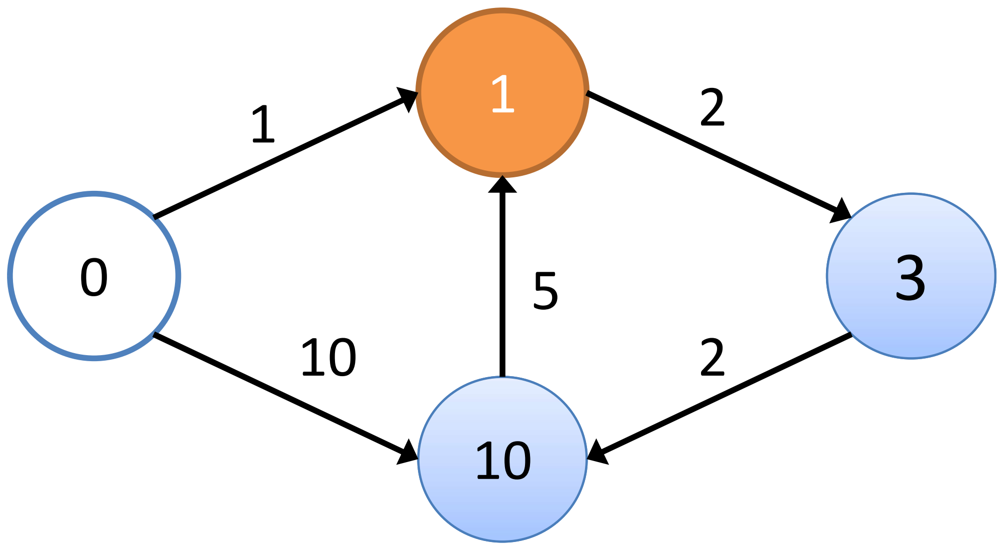


Pop next node from queue

Q={1,10,inf}

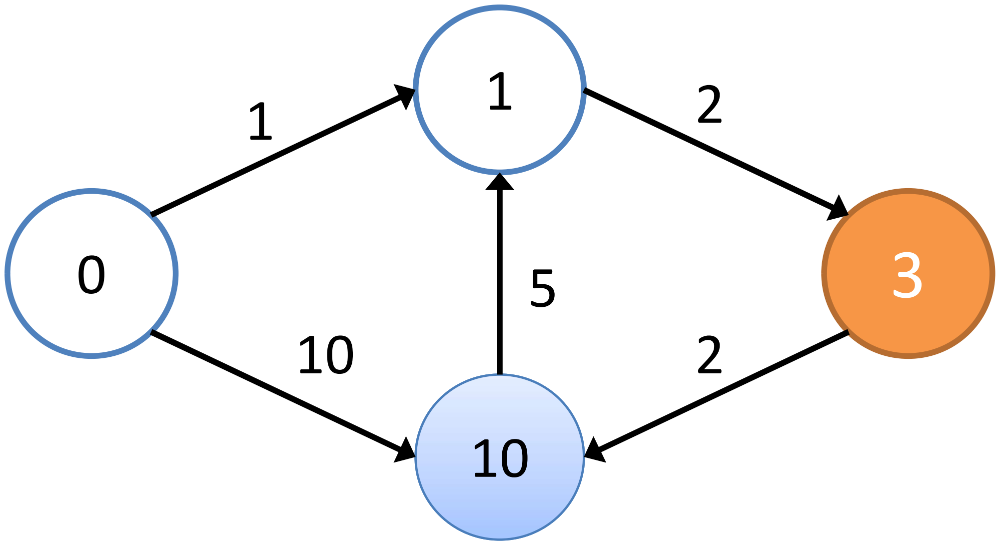


Update distances

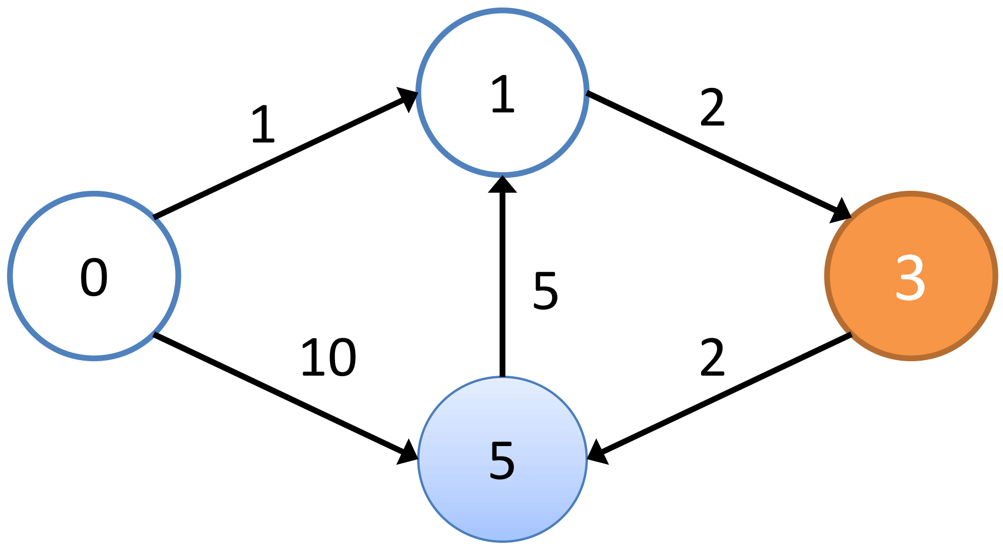


Pop next node from queue

Q={3,10}

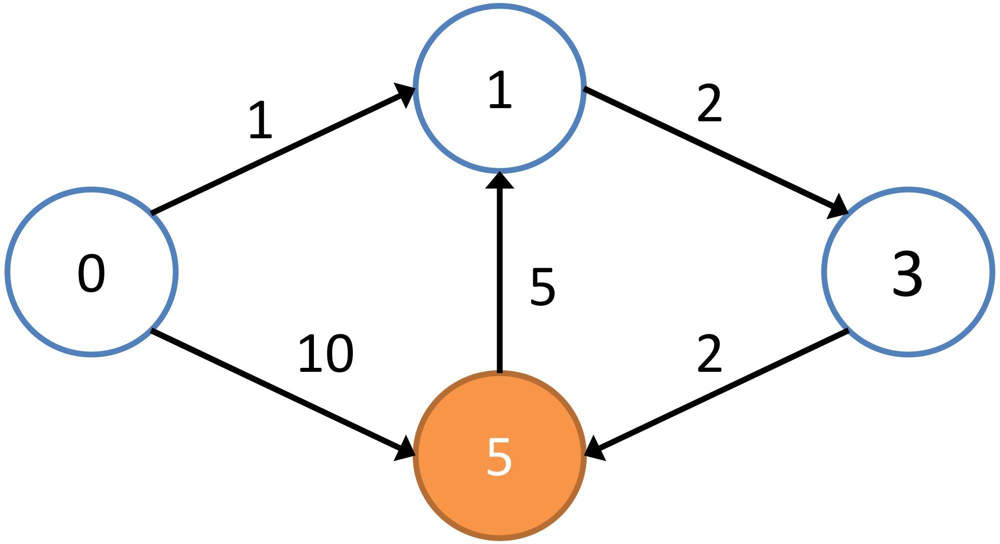


Update distances

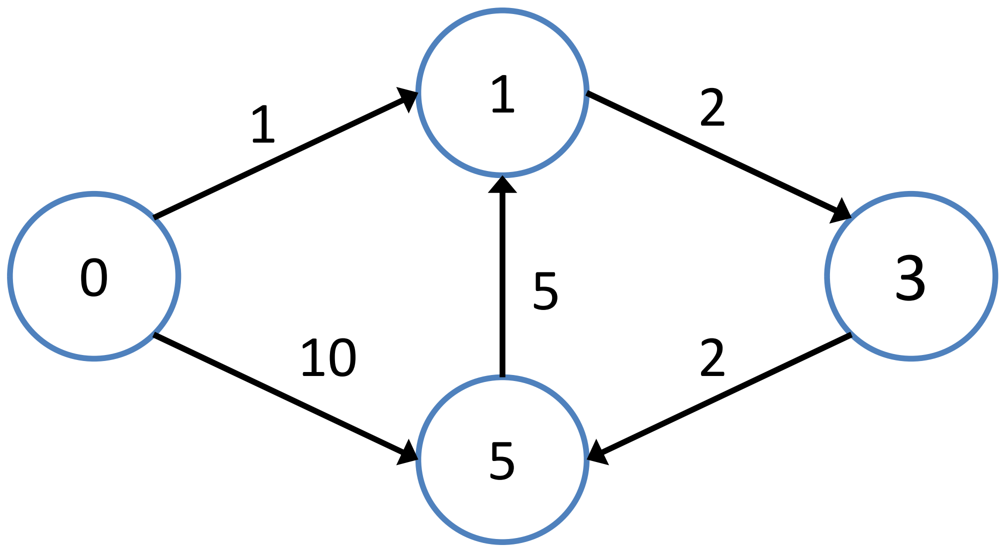


Pop last node, finished!

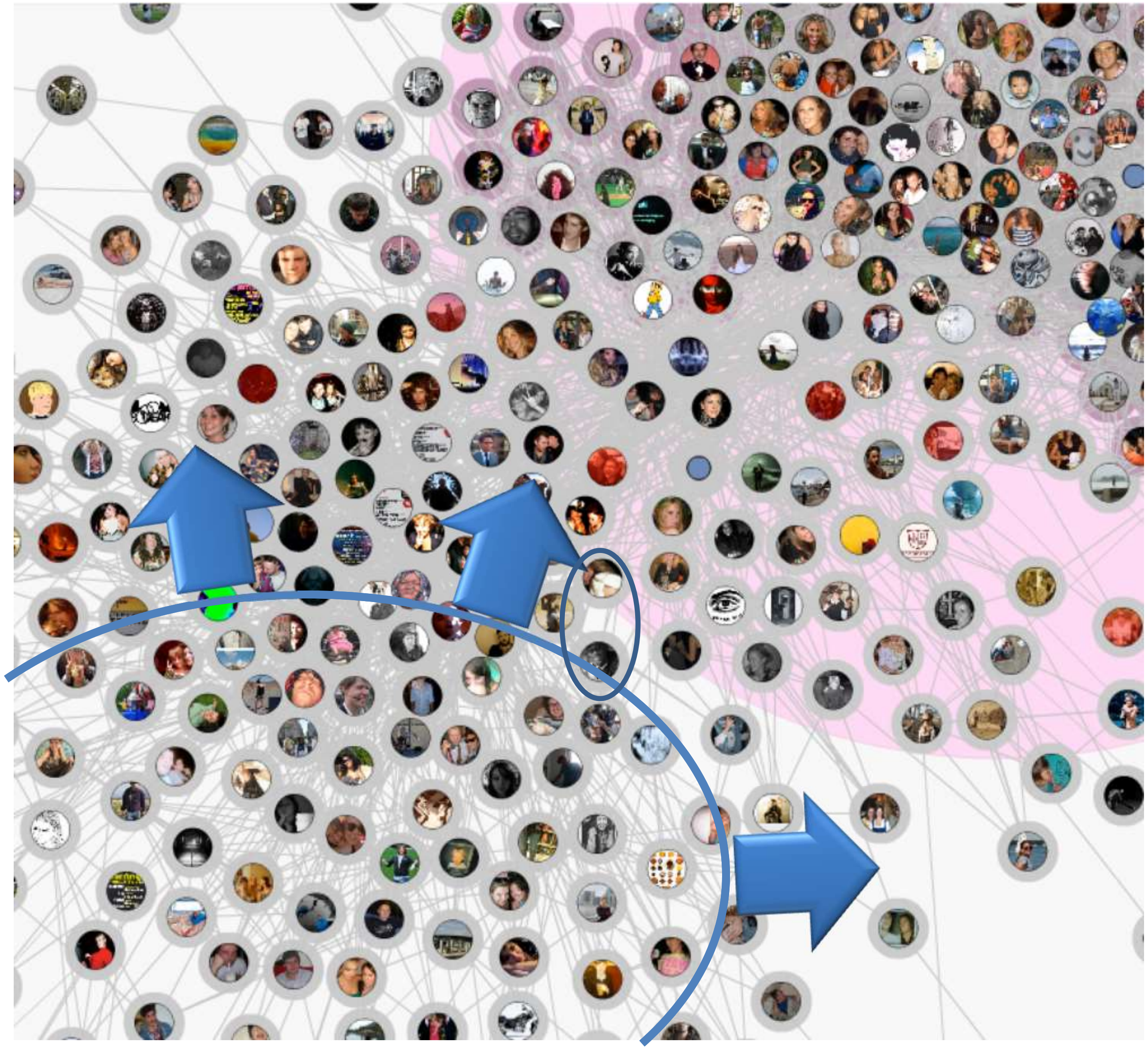
Q={5}



Computed distances

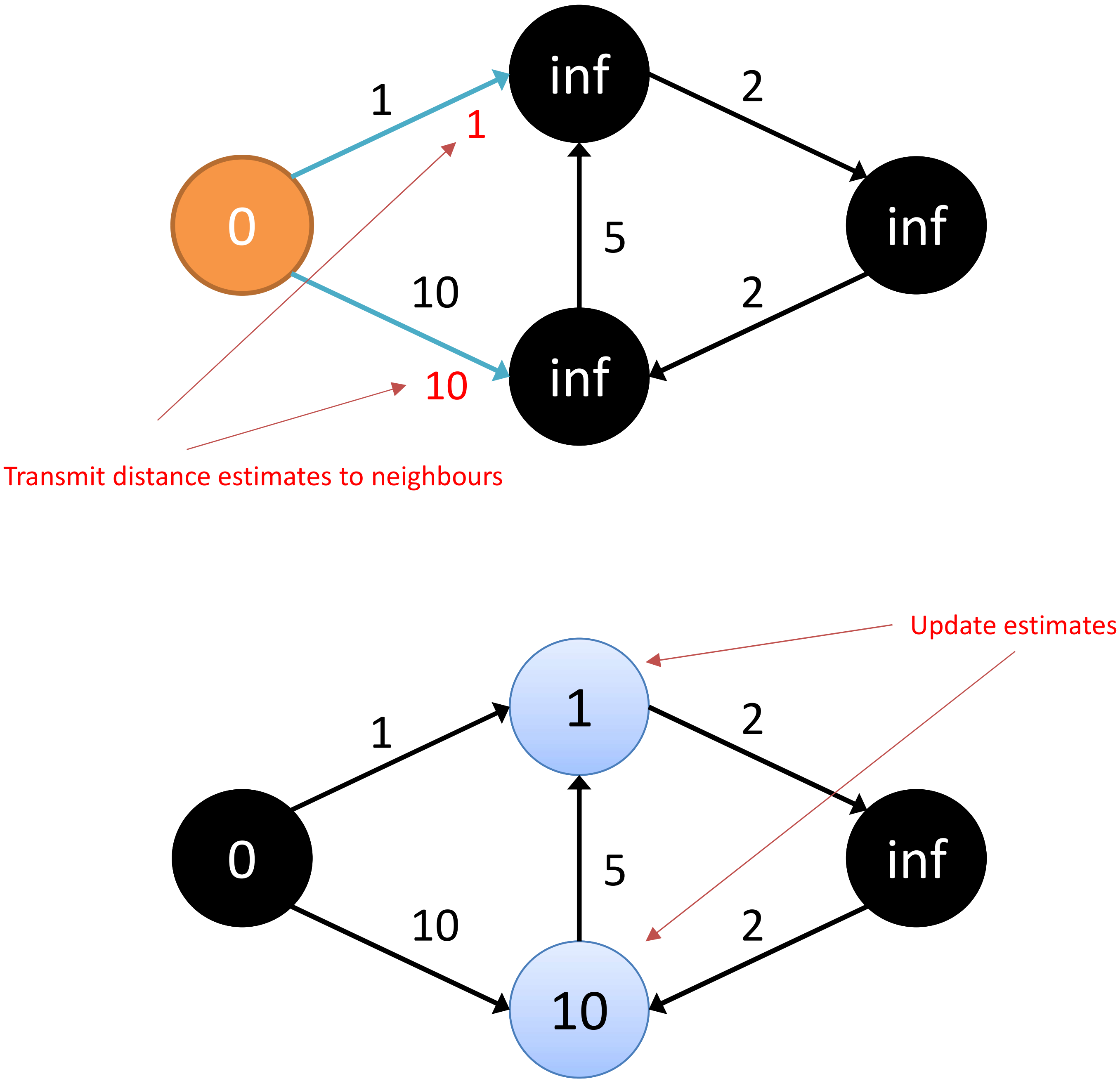


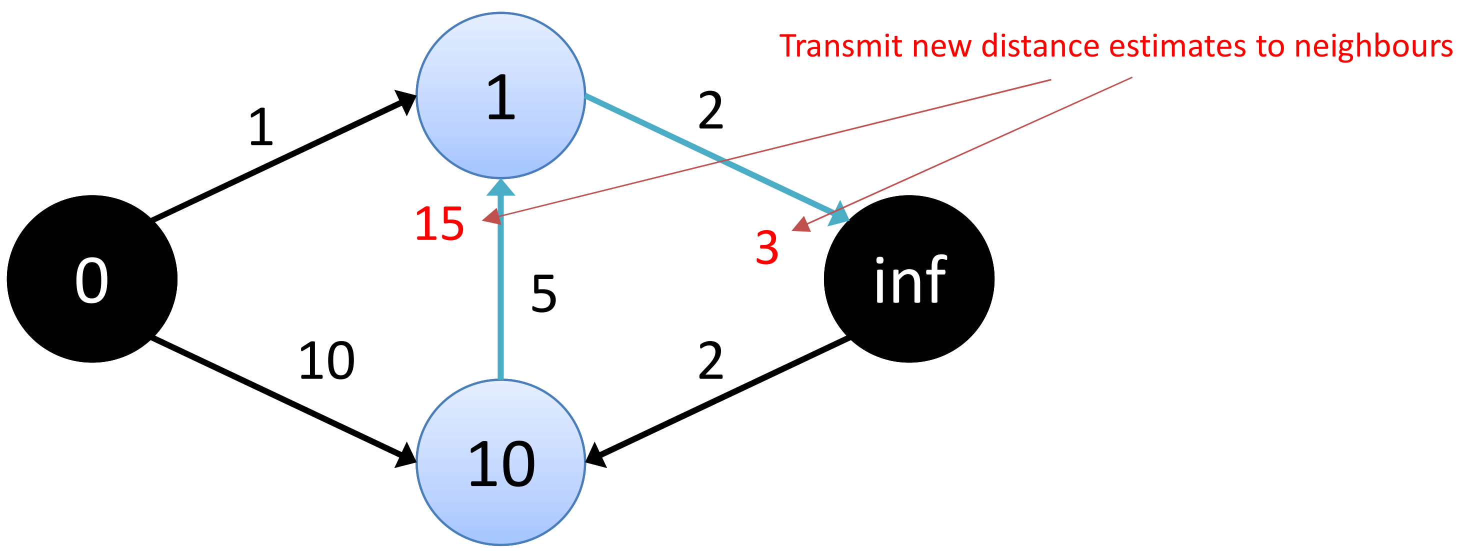
Dijkstra on a billion nodes graph

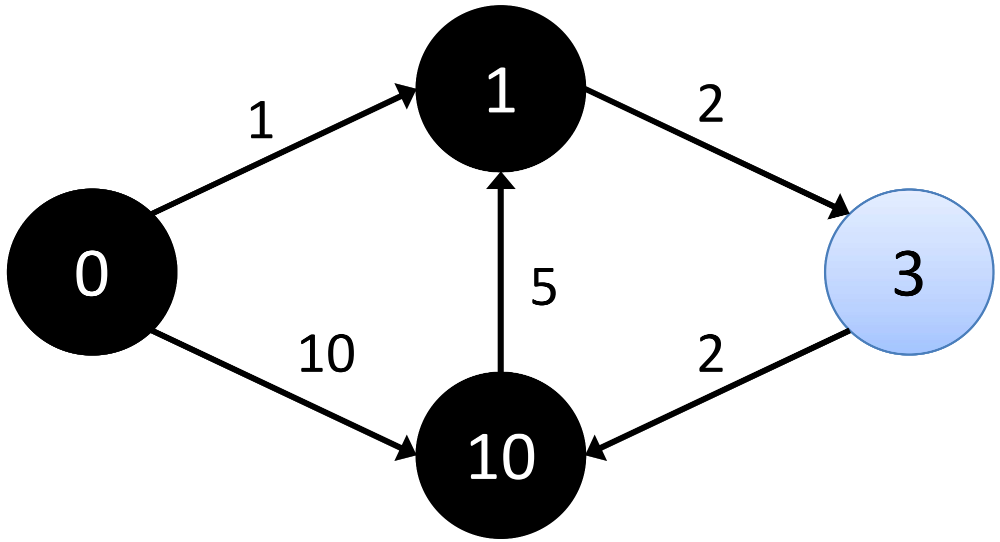


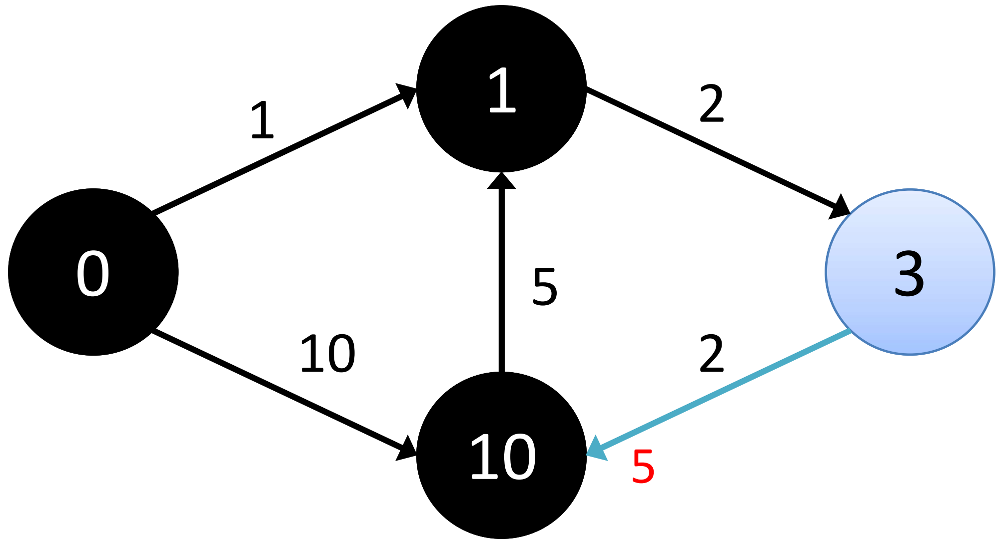
Parallel Breadth-First Search (PBFS)

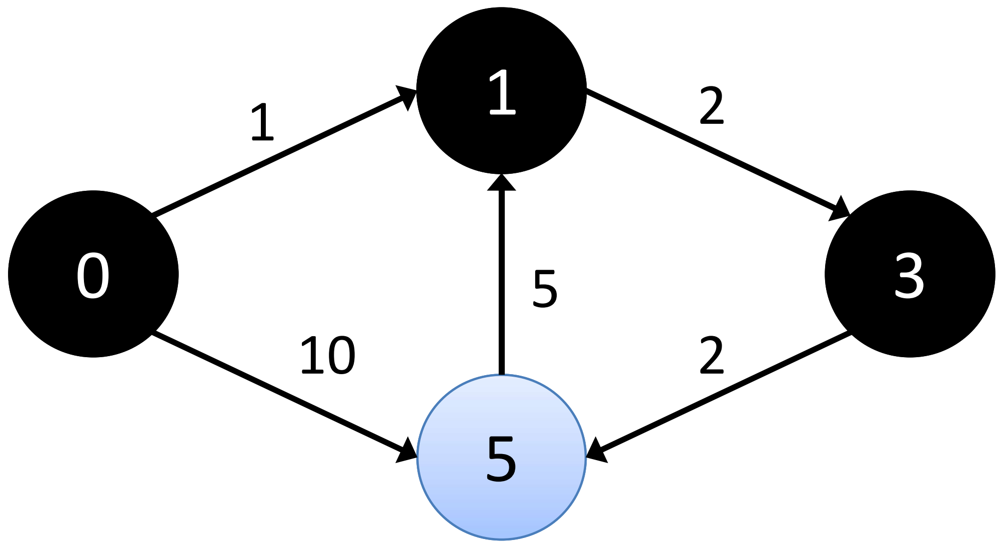
- Each node maintains current distance estimate
- Upon receive of a message from neighbors update estimate
  - If newly computed distance is shorter, inform neighbors

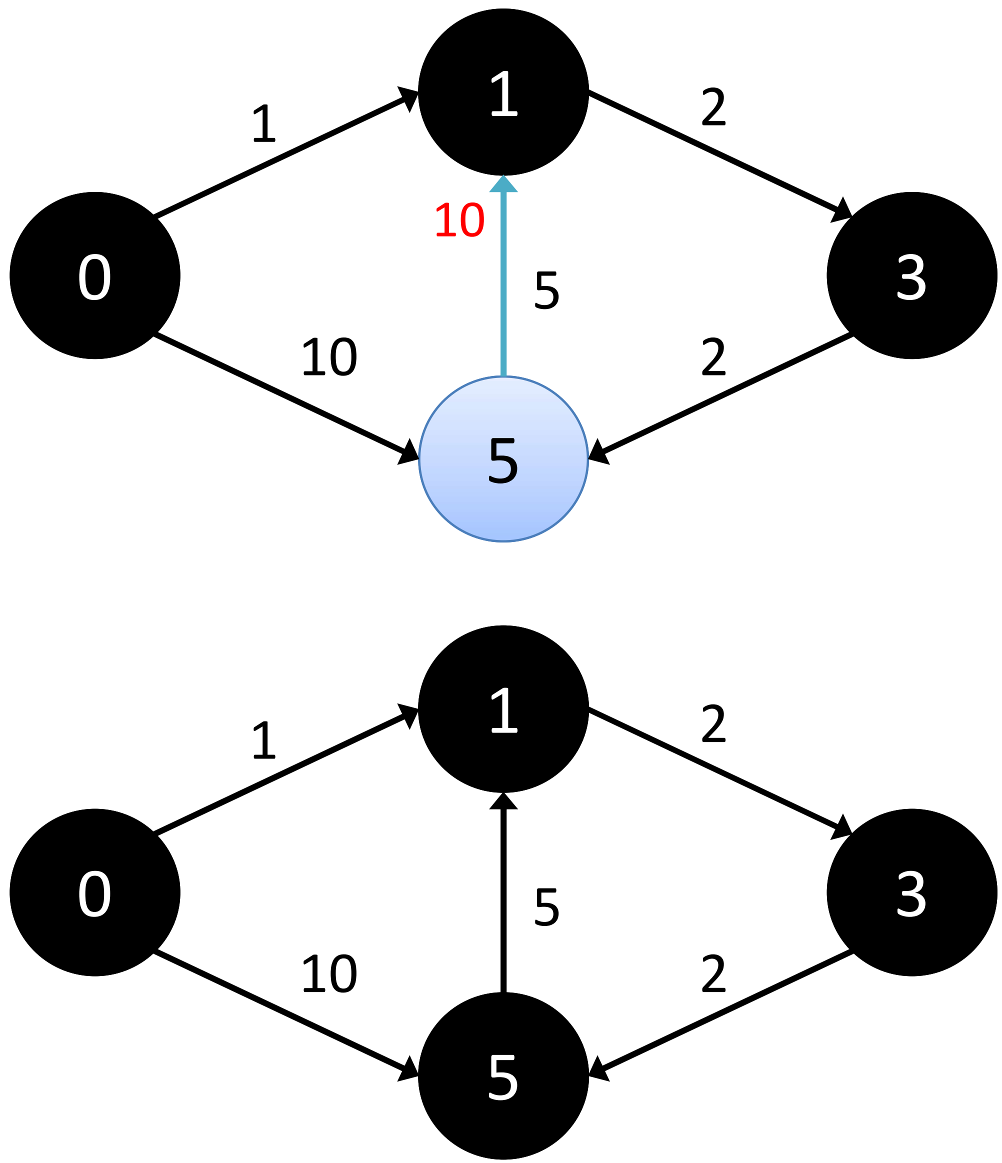

inf
1
2
1
0
5
inf
10
2
10
inf
Transmit distance estimates to neighbours
Update estimates
1
1
2
0
5
inf
10
2
10


Transmit new distance estimates to neighbours
1
2
1
15
3
0
5
inf
10
2
10


1
2
1
0
5
3
10
2
10


1
2
1
0
5
3
10
2
10
5


1
1
2
0
5
3
10
2
5



0
1
3
5
1
2
10
5
2
10
0
1
3
5
1
2
5
10
2

PBFS vs Dijkstra

 PBFS: More (redundant) computations of distances until true shortest path is found

BUT

 Many parallel calculations per clock tick. No need of a global priority query, only local state maintained at each node

Shortest Path Code

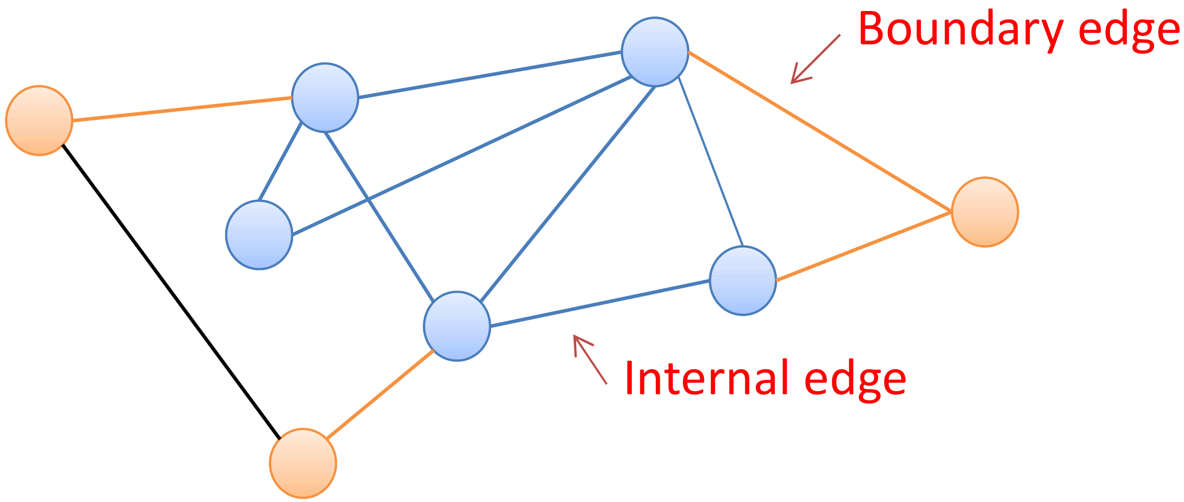
```
class ShortestPathVertex
    : public Vertex<int, int, int> {
  void Compute(MessageIterator* msgs) {
    int mindist = IsSource(vertex_id()) ? 0 : INF;
    for (; !msgs->Done(); msgs->Next())
      mindist = min(mindist, msgs->Value());
    if (mindist < GetValue()) {
      *MutableValue() = mindist;
      OutEdgeIterator iter = GetOutEdgeIterator();
      for (; !iter.Done(); iter.Next())
        SendMessageTo(iter.Target(),
                      mindist + iter.GetValue());
    }
    VoteToHalt();
  }
};
```

PageRank Code

```
class PageRankVertex
    : public Vertex<double, void, double> {
 public:
  virtual void Compute(MessageIterator* msgs) {
    if (superstep() >= 1) {
      double sum = 0;
      for (; !msgs->Done(); msgs->Next())
        sum += msgs->Value();
      *MutableValue() =
          0.15 / NumVertices() + 0.85 * sum;
    }

    if (superstep() < 30) {
      const int64 n = GetOutEdgeIterator().size();
      SendMessageToAllNeighbors(GetValue() / n);
    } else {
      VoteToHalt();
    }
  }
};
```

Semi-clustering in a social graph

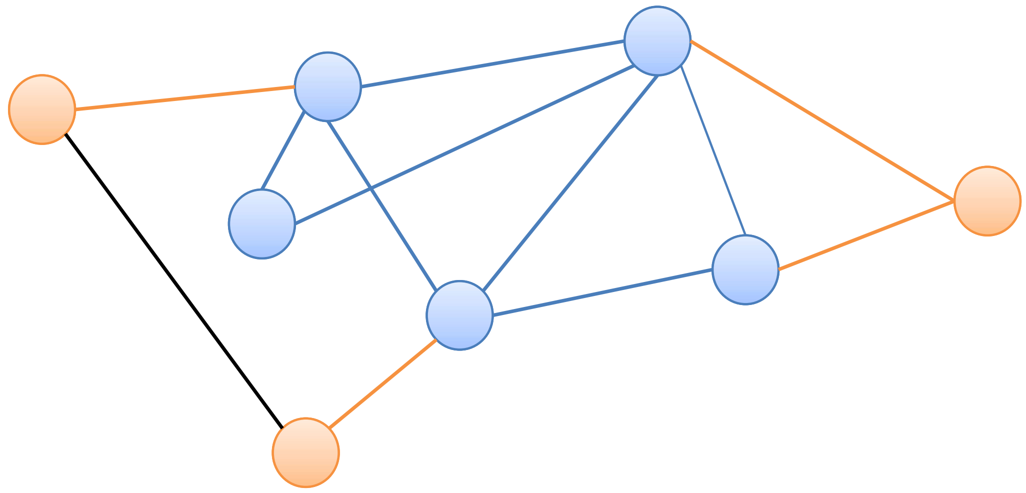
- A semi-cluster in a social graph is a group of people who interact frequently with each other and less frequently with others.
  - A person may belong to multiple semi-clusters



Evaluation of Semi-clusters

- I_c: sum of weights of internal edges
- B_c: sum of weights of boundary edges
- V_c: size of semi-cluster
- F_b: boundary edge score factor (0..1)

$$S_c = \frac{I_c - f_B B_c}{V_c(V_c - 1)/2}$$



$I_c = 7$ $B_c = 4$ $V_c = 5$

Computing Semi-clusters in Pregel

- Each vertex maintains a list containing at most C_{max} semi-clusters, sorted by score.
- In super-step 0 each node creates its own cluster and informs neighbors.
- In subsequent super-steps a vertex V iterates over the semi-clusters sent to it on the previous super-step.
  - If a semi-cluster does not already contain V and is not full then V is added to that cluster
  - The best k semi-clusters (sorted by their scores) are sent to neighbors
  - Node keeps a list of semi-clusters that contain V (itself)
- Stop if no new semi-clusters are formed of after a set of iterations