2014.11.6

Cloudera World Tokyo 2014

Apache Spark GraphX 入門

三菱UFJインフォメーションテクノロジー株式会社
ITプロデュース部
土佐鉄平

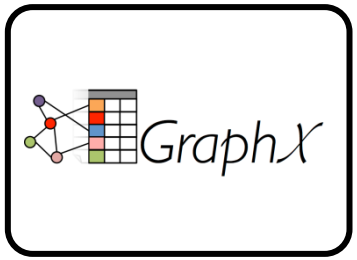
Overview

1. 当社ご紹介
2. GraphXとは
3. グラフ構造データとは
4. グラフ構造データ分析とは
5. GraphXの基本操作
6. GraphXのPregel処理
7. まとめ

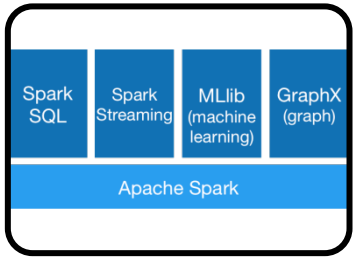
私たちは、三菱東京UFJ銀行をはじめとする
三菱UFJフィナンシャル・グループの総合金融サービスを支える
金融×ITのリーディングカンパニーです。

2．GraphX とは

Apache Spark のコンポーネントのひとつ



大容量のグラフデータを並列分散環境で
処理するためのフレームワーク



ETL処理やSQL処理、機械学習処理と合わせ
Spark上でひとつのシステムで実装可能

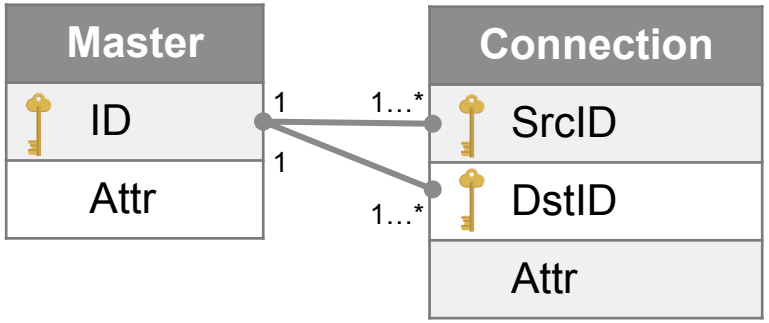
2．GraphX とは - 他技術比較

	Giraph	GraphLab	GraphX
スピード	遅め	速い	速め
安定度	安定	新しい	新しい
言語	Java	Python	Scala
ETL連動	困難	容易	容易
ライブラリ	不足	充実	不足

一長一短あるが Spark エコシステムとしての GraphX は魅力

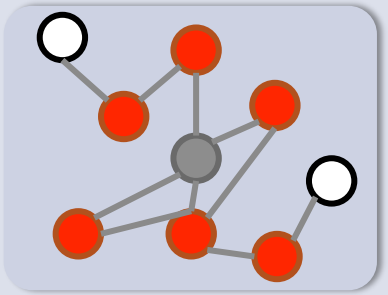
3．グラフ構造データとは

- 「点（Vertex）」と「辺（Edge）」からなるデータ
- マスタデータとマスタ間の関係性を持つデータがあれば、多くの場合でグラフデータとして表現可能（下図参照）
  - 例）
    - ✓ フォロー（人：点、フォロー：辺）
    - ✓ メール（人：点、メール：辺）
    - ✓ 路線（駅：点、路線：辺）
    - ✓ 道路（交差点：点、道：辺）
    - ✓ 購入履歴（人、商品：点、履歴：辺）
    - ✓ 口座振込（口座：点、振込：辺）
    - ✓ CRM（案件、担当：点、関わり：辺）
  - 等々



身の回りのサービス・業務システム上の様々なデータをグラフデータとして表現可能

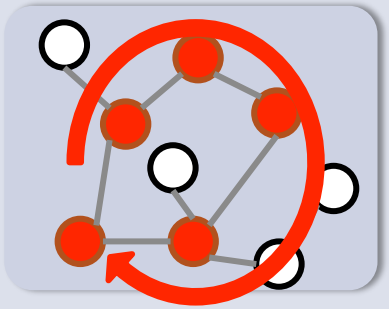
4．グラフ構造データ分析とは



繋がりの抽出

繋がりあっている点と点を抽出

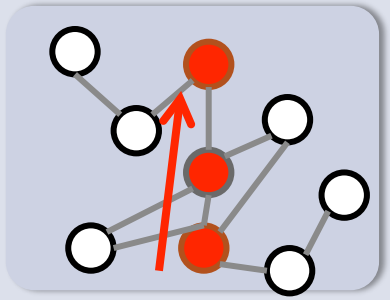
例：Aさんと２階層以内につながっている友達を抽出する



輪の抽出

グラフの中から循環構造になっている点の抽出

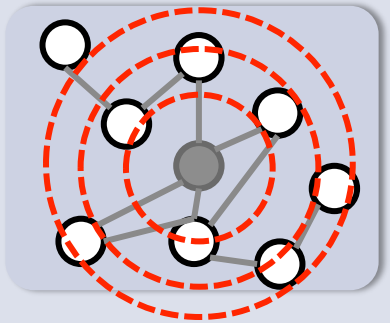
例：Aさんが所属する友達の輪を抽出



距離の計測

点と点の間の距離を計測

例：AさんからBさんまで何次の隔たりになっているか計測



影響の計測

点と点の間の影響度合いの計測

例：Aさんの影響はその友達にどの程度与えるのか計測

表構造で扱っていてはできない分析が可能

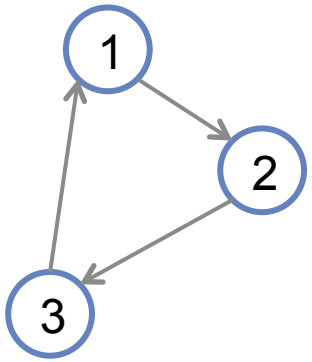
5．GraphX 基本操作

- Edgeリストからのグラフ生成

```
val graph = GraphLoader.edgeListFile(sc, "edges.tsv")
```

- Edgeリストファイル

```
1 2
2 3
3 1
```



5．GraphX 基本操作

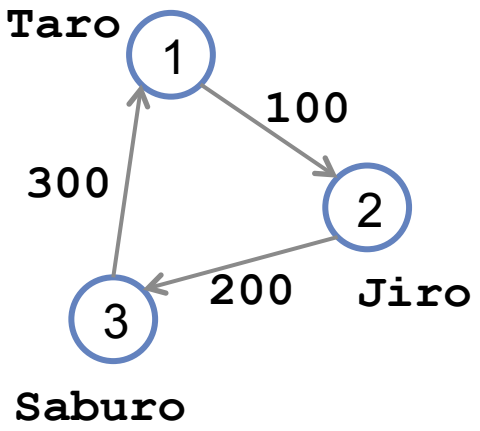
- マスタデータとコネクションデータからグラフ生成

master.csv

```
1, Taro
2, Jiro
3, Saburo
```

connections.csv

```
1, 2, 100
2, 3, 200
3, 1, 300
```



5．GraphX 基本操作

- マスタデータとコネクションデータからグラフ生成

```
// 元データを読み込み
val master = sc.textFile("master.csv")
val conn = sc.textFile("connection.csv")

// 頂点データを生成
val vertices:RDD[(VertexId, String)]
        = master.map( line => {
                val cols = line.split(",")
                (cols(0).toLong, cols(1))})

// 辺データを生成
val edges:RDD[Edge[Long]]
        = conn.map( line => {
                val cols = line.split(",")
                Edge(cols(0).toLong, cols(1).toLong, cols(2).toLong)})

// 頂点データと辺データからグラフを生成
val graph:Graph[String, Long] = Graph(vertices, edges)
```

5．GraphX 基本操作

- グラフデータの中身を標準出力

```
graph.vertices.collect.foreach(println(_))

// (1, Taro)
// (2, Jiro)
// (3, Saburo)


graph.edges.collect.foreach(println(_))

// Edge(1, 2, 100)
// Edge(2, 3, 200)
// Edge(3, 1, 300)
```

5．GraphX 基本操作

- Taro は誰をフォローしているかを取得

```
graph
  // Out方向の隣接頂点のリストを取得
  .collectNeighbors(EdgeDirection.Out)
  // :List[(VertexId, List[(VertexId, String)]]
  // リストの中から VertexId が 1 のものをフィルタ
  .filter(v => v._1 == 1)
  // 1行だけがフィルタされているはずなので1行目を取得
  .first
// _1にはVertexId、_2に隣接頂点リストが入っている
  ._2
  // 隣接頂点リストを標準出力
  .foreach(println(_))

// (2, Jiro)
```

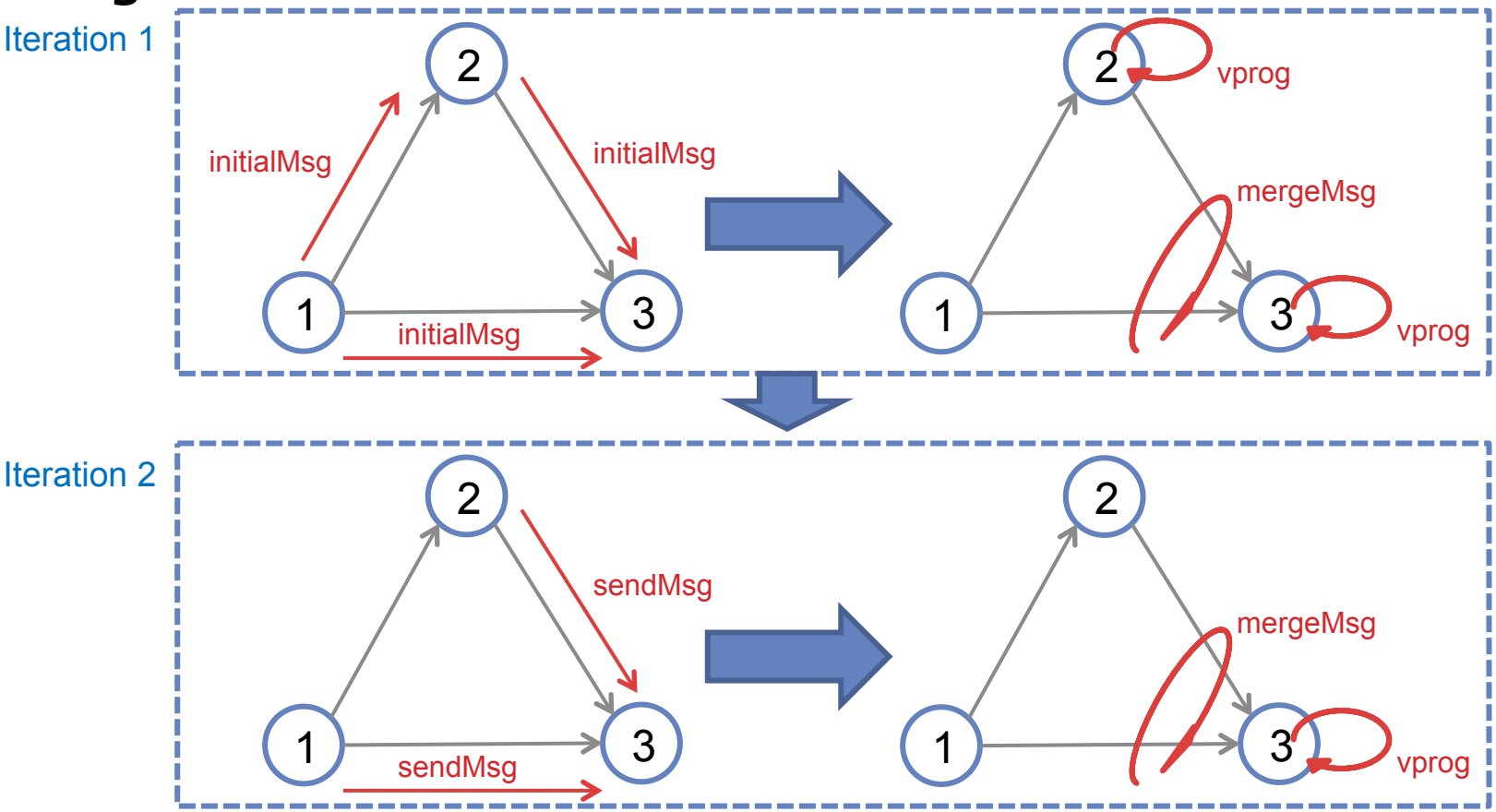
6．GraphX の Pregel 処理

- **Pregelとは**
  - Pregel は Google が開発したグラフアルゴリズムを実装するためのスケーラブルなプラットフォーム実装
  - Pregel のソースコード自体は公開されておらず、論文が発表されているのみ
  - GraphX は、この論文の内容を実装した関数を用意しており、GraphX でこれを利用可能

Pregelによって複雑なグラフ分析処理を
並列分散処理として実装可能

6．GraphX の Pregel 処理

- **Pregel 処理概要**



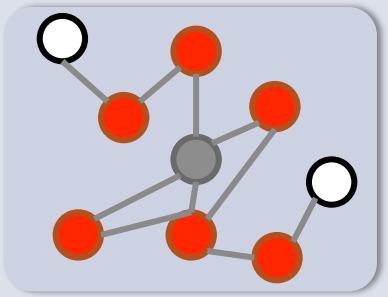
※　Iteration終了条件：指定回数行われる　Or　送信Msgが無くなる（受信した頂点しか送信しない）

6．GraphX の Pregel 処理

- **Pregel API**

```
Pregel(
        // 処理対象のグラフデータ
        graph,
        // 最初のいてレーションで送信するメッセージ
        initialMsg:A,
        // 最大何回イテレーションするか
        maxIter:Int,
        // どちら方向にメッセージ送信するか
        activeDir:EdgeDirection)
(
        // イテレーション毎に頂点で稼動させる処理（Aは受信したメッセージ）
        vprog: (VertexId, VD, A) => VD,
        // イテレーション毎に辺上に何を流すか決定する処理
        sendMsg: (EdgeTriplet[VD, ED]) => Iterator[(VertexId, A)],
        // 複数のEdgeからメッセージを受信した時の処理
        mergeMsg: (A, A) => A
// 処理結果として新しいGraphデータを返す
) : Graph[VD, ED]
```

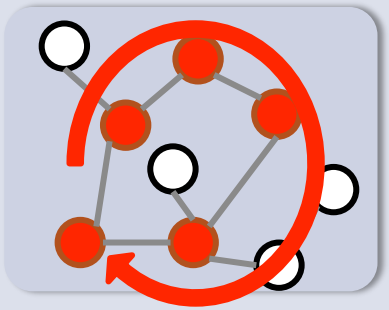
6．GraphX の Pregel 処理



①繋がりの抽出

繋がりあっている点と点を抽出

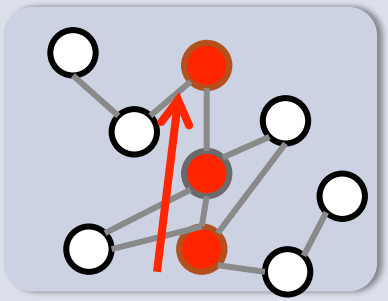
例：Aさんと２階層以内につながっている友達を抽出する



②輪の抽出

グラフの中から循環構造になっている点の抽出

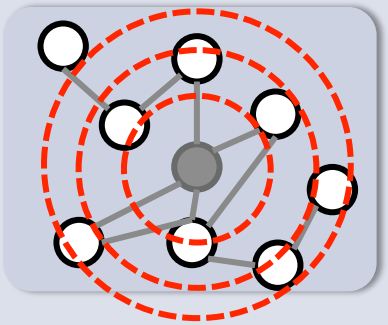
例：Aさんが所属する友達の輪を抽出



③距離の計測

点と点の間の距離を計測

例：AさんからBさんまで何次の隔たりになっているか計測



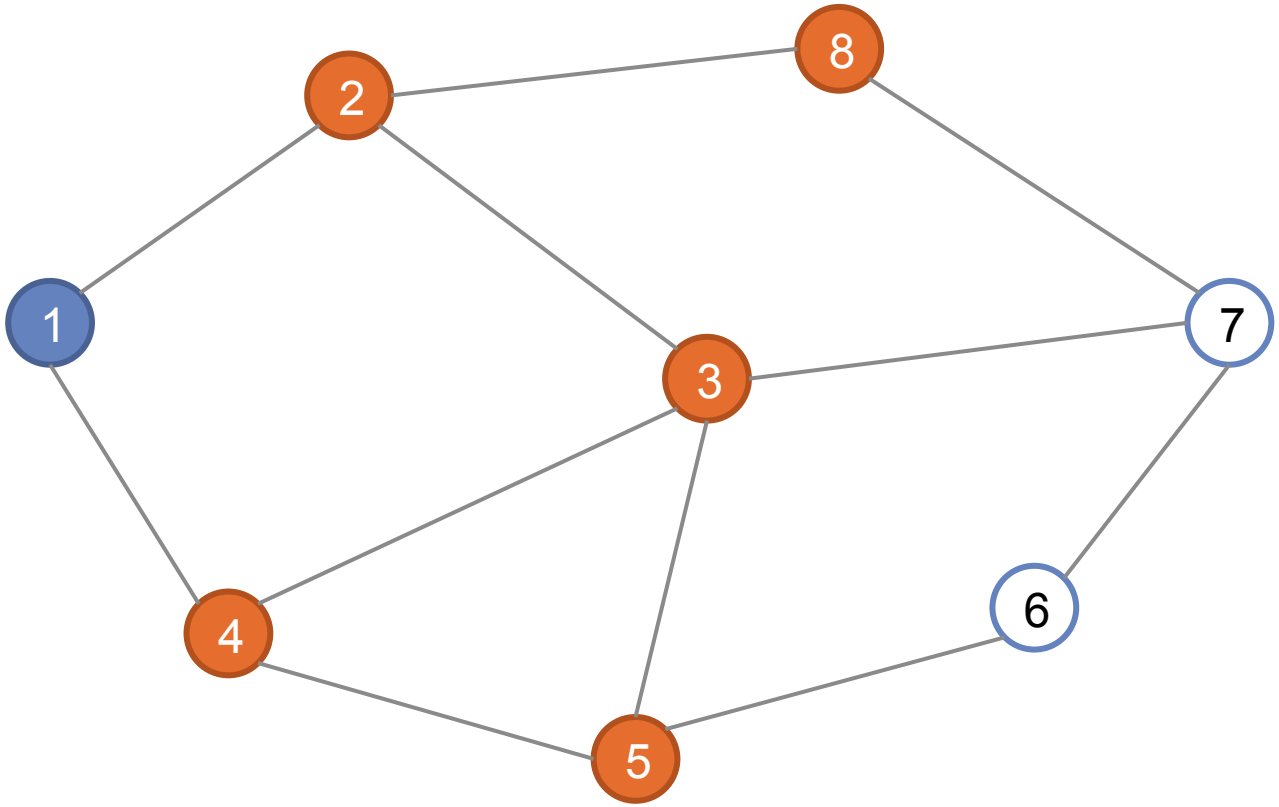
④影響の計測

点と点の間の影響度合いの計測

例：Aさんの影響はその友達にどの程度与えるのか計測

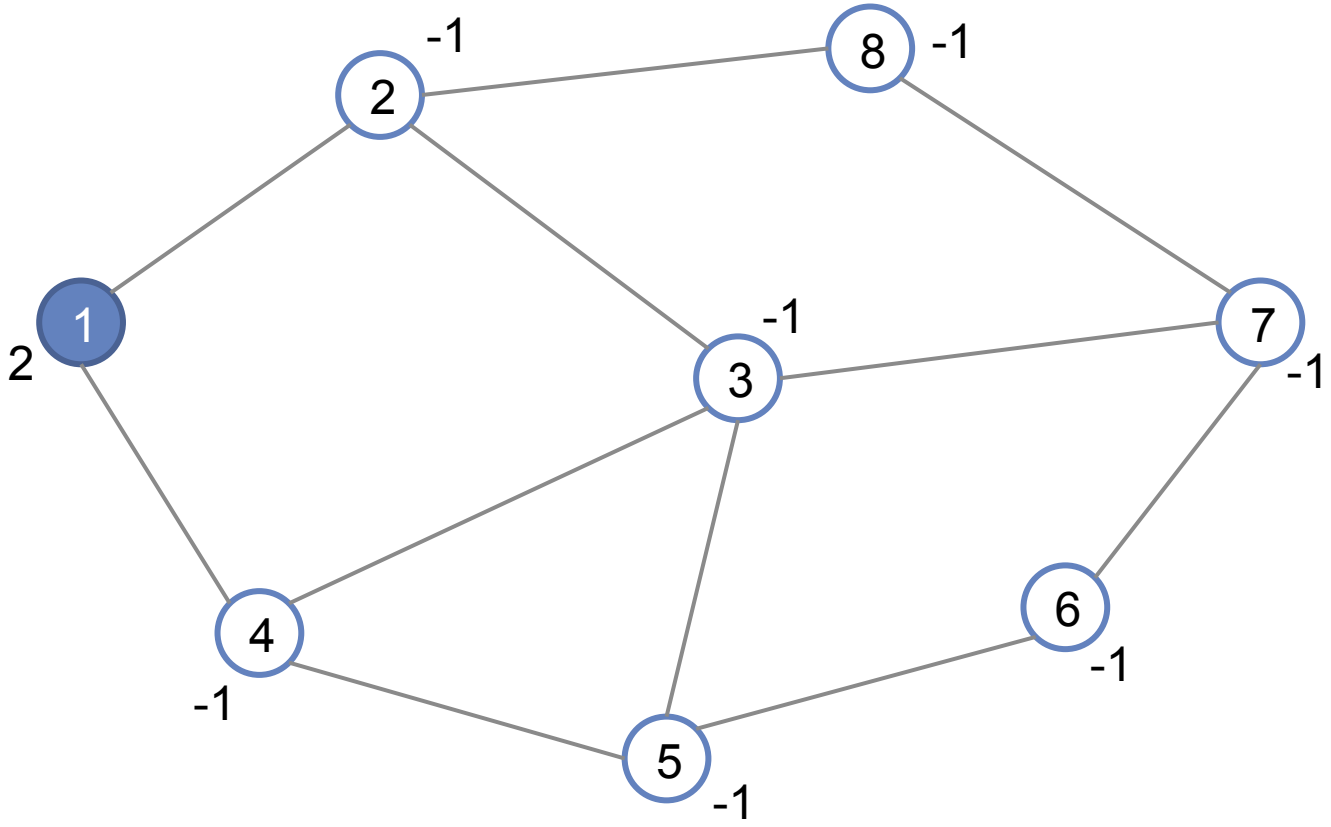
6．GraphX の Pregel 処理
①　繋がりの抽出

2階層以内の友達を抽出



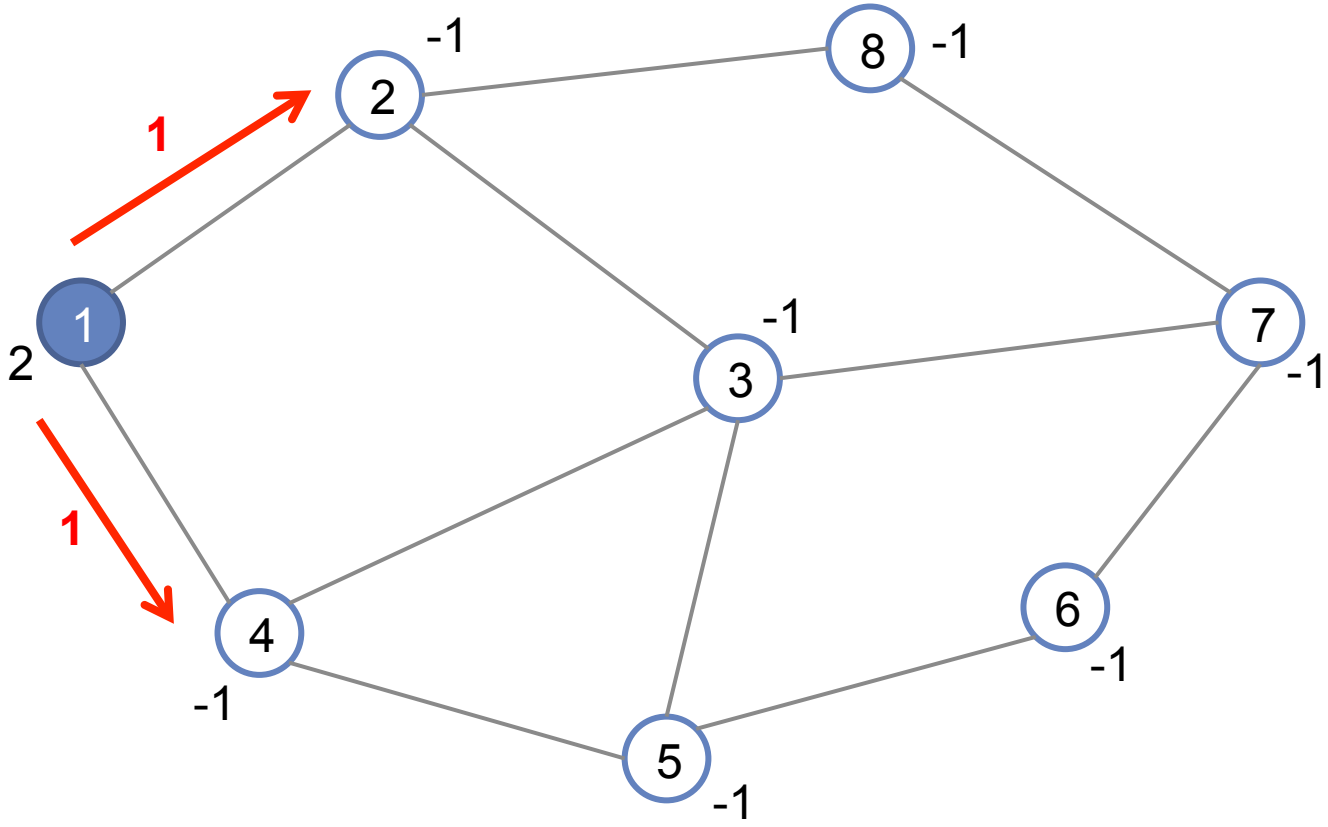
6．GraphX の Pregel 処理
①　繋がりの抽出

**【事前準備】1番に指定階層の２を持たせて、
その他頂点に-1を持たせる**



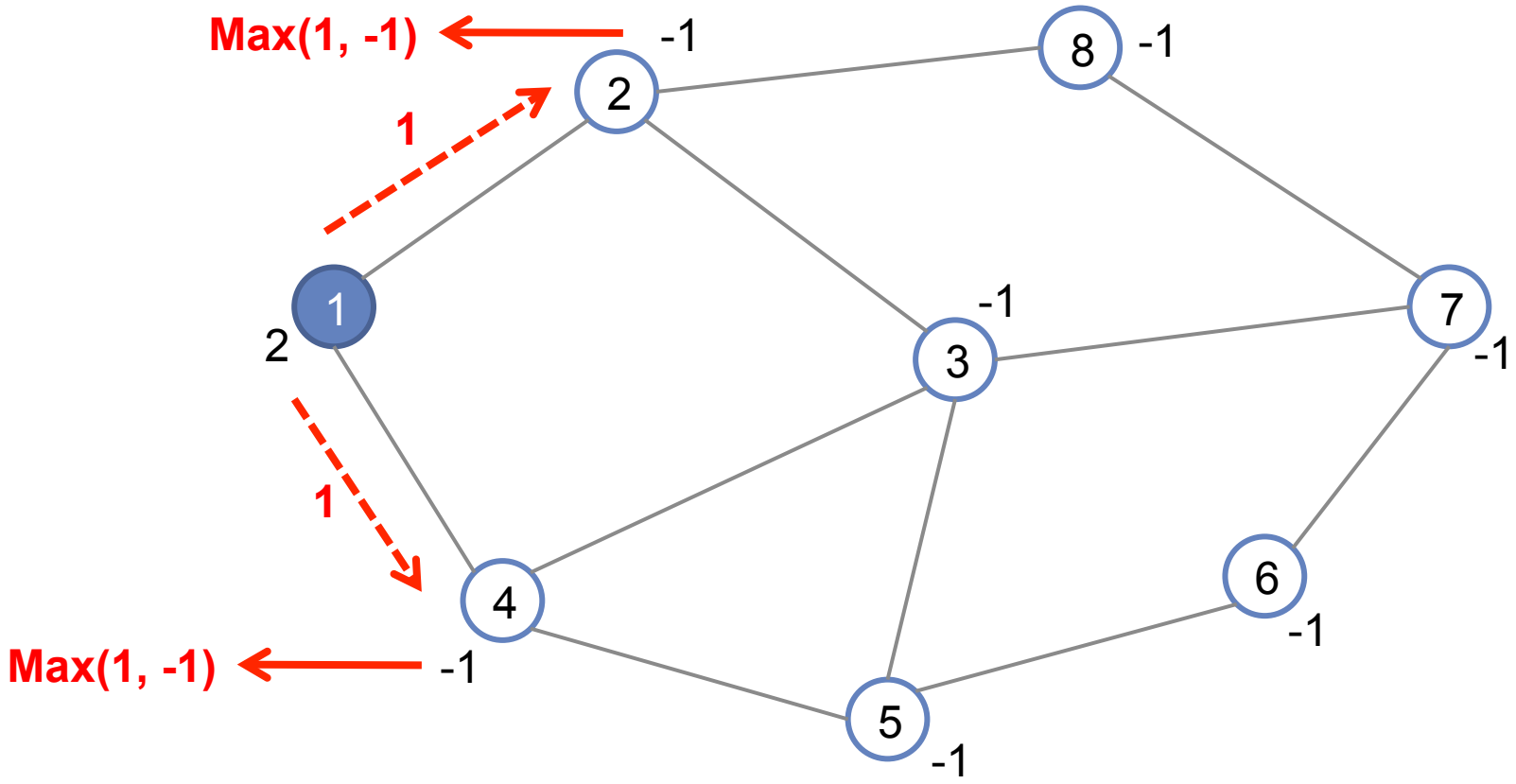
6．GraphX の Pregel 処理
①　繋がりの抽出

【Iteration１】片方が0以下で、且つ片方が０より大きい時、0より大きい方が１減算して、もう片方に送信する。両方とも0以下なら何も送信しない



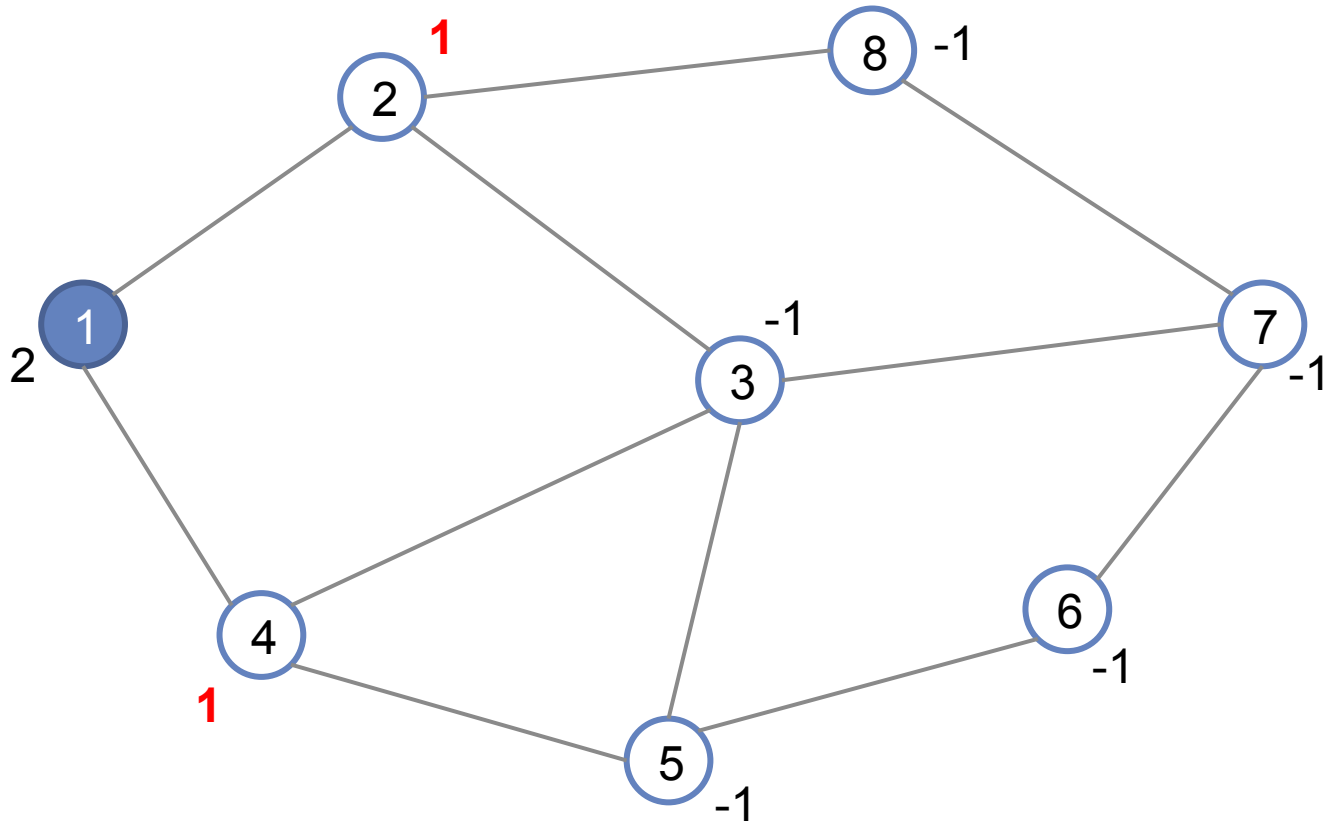
6．GraphX の Pregel 処理
①　繋がりの抽出

【Iteration１】受信した数値と、自分の数値を比較して、大きい方の値を自分の値に設定する



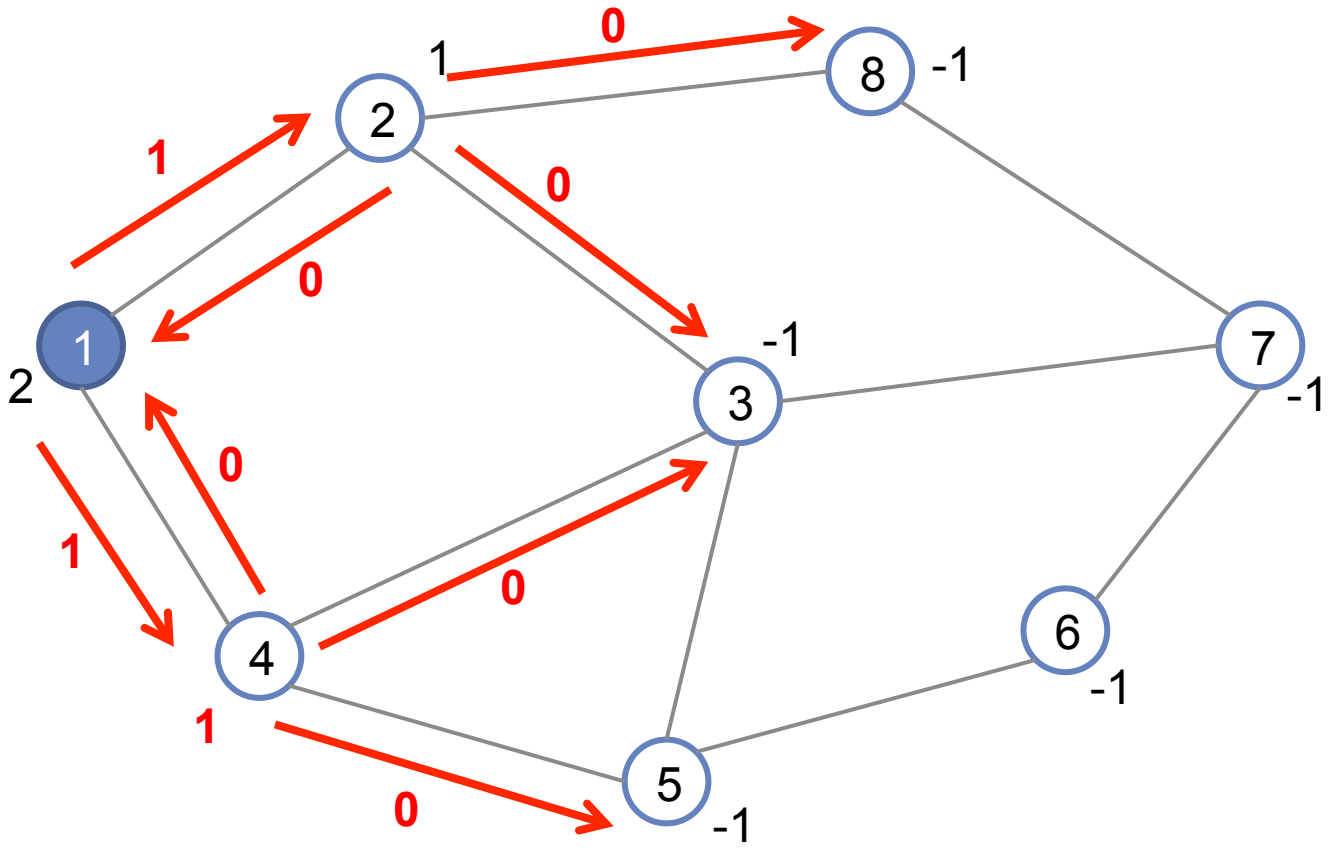
6．GraphX の Pregel 処理
①　繋がりの抽出

【Iteration１】完了



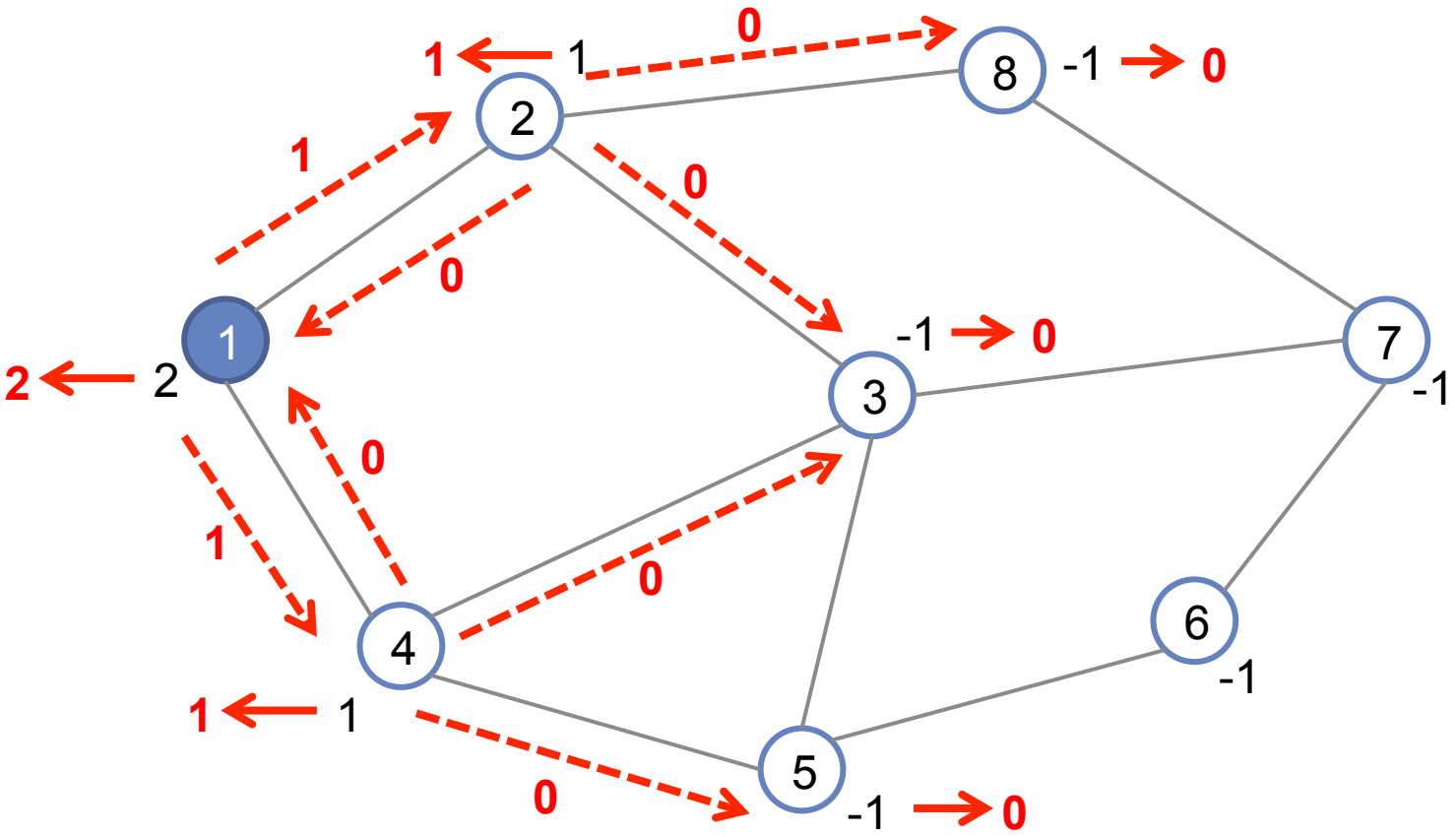
6．GraphX の Pregel 処理
①　繋がりの抽出

【Iteration２】先ほどと同様の処理を繰り返す。両方とも０より大きかったら、双方１を減算して送りあう。



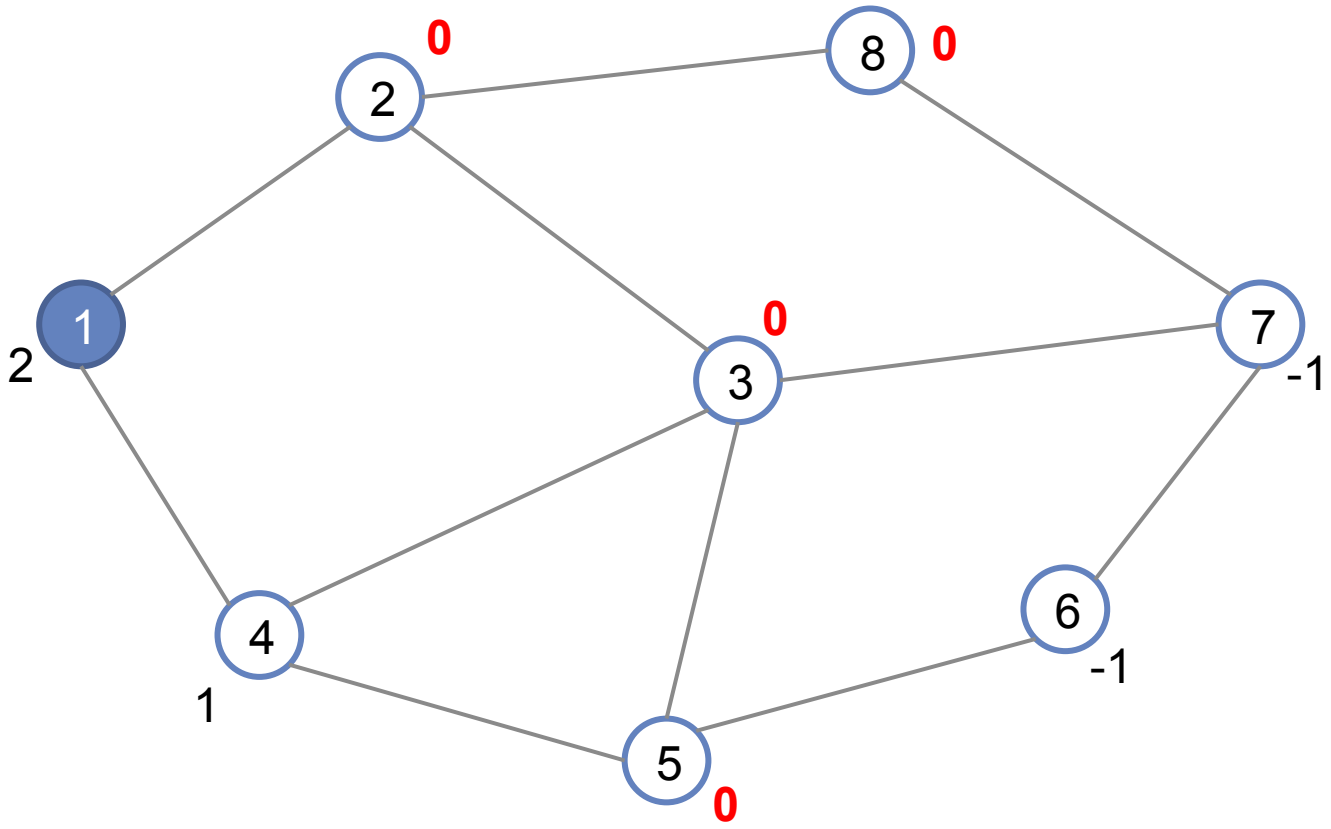
6．GraphX の Pregel 処理
①　繋がりの抽出

【Iteration２】受信した数値と、自分の数値を比較して、大きい方の値を自分の値に設定する



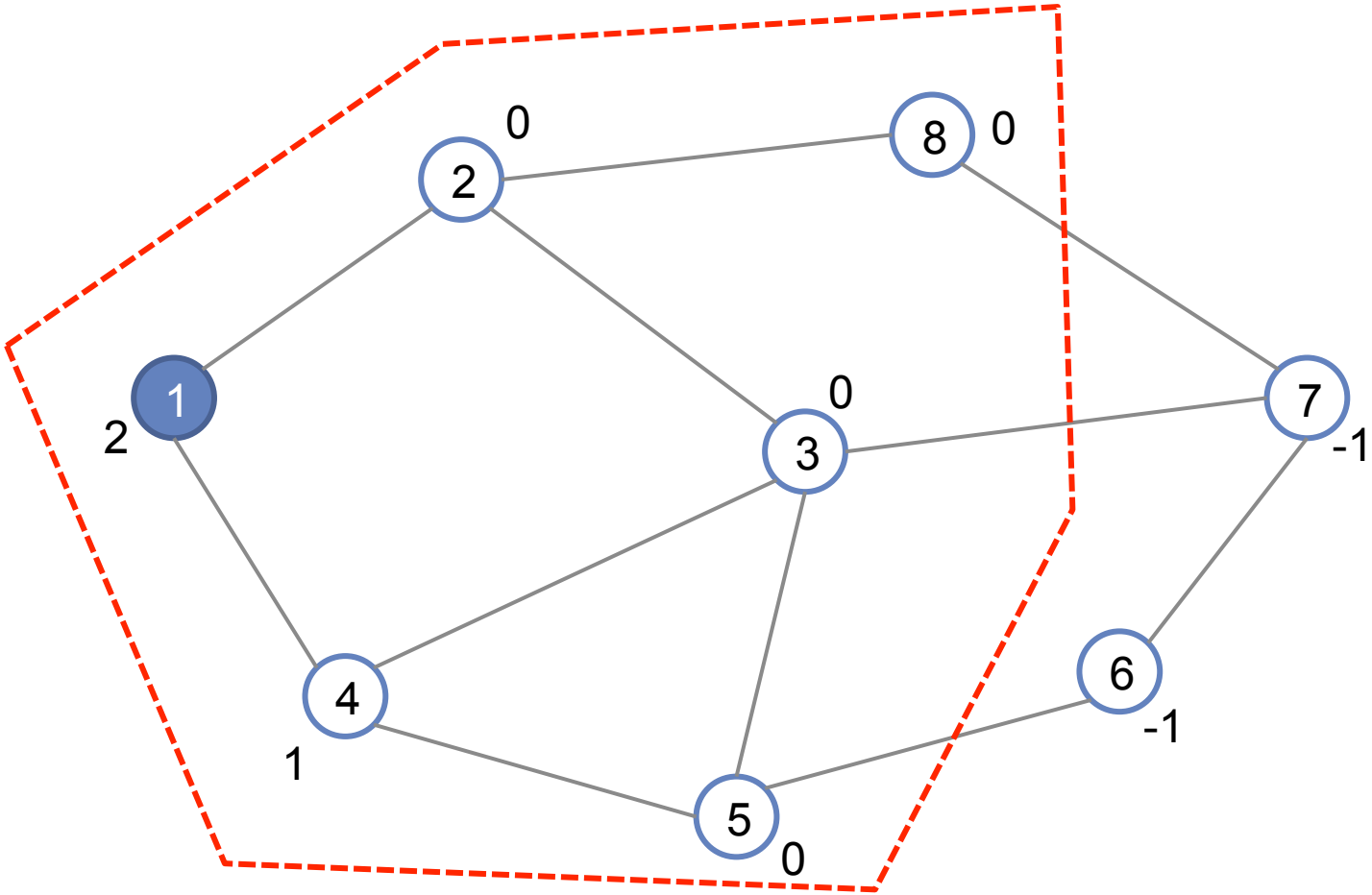
6．GraphX の Pregel 処理
①　繋がりの抽出

【Iteration2】完了



6．GraphX の Pregel 処理
①　繋がりの抽出

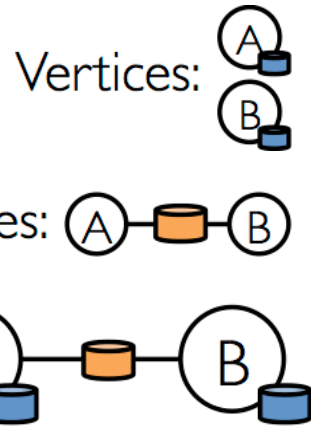
【Pregel後】0以上の値を持つ頂点を抽出



6．GraphX の Pregel 処理
①　繋がりの抽出

```
val friends = Pregel(
 graph.mapVertices((vid, value)=> if(vid == 1) 2 else -1),
 // 最初のイテレーションで送信するメッセージ
 -1,
 // 指定階層値の回数イテレーションする
 2,
 // 双方向にメッセージ送信する
 EdgeDirection.Either)
( // 受信した値と自分の値のうち、大きい方を設定する
  vprog: (vid, attr, msg) => math.max(attr, msg),
  // イテレーション毎に辺上に何を流すか決定する処理（後述）
  sendMsgFunc,
  // 複数のEdgeから値を受信したら大きい方をとる
  (a, b) => math.max(a, b)
)
// 頂点の値が0以上の頂点を抽出
.subgraph(vpred = (vid, v) = v >= 0)
```

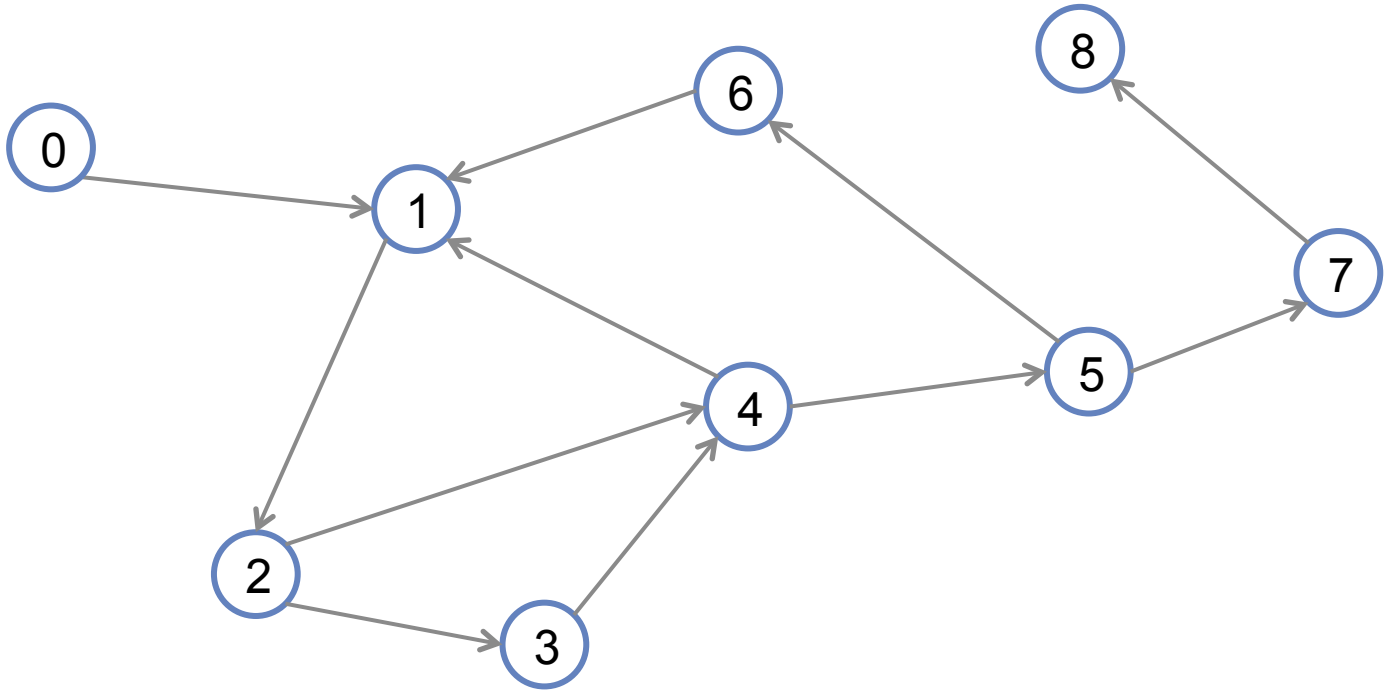
6．GraphX の Pregel 処理
①　繋がりの抽出




```
def sendMsgFunc(edge:EdgeTriplet[Long, Long]) = {
 if(edge.srcAttr <= 0){
  if(edge.dstAttr <= 0){
   // 両方とも0以下なら何も送信しない
   Iterator.empty
  }else{
   // 片方0以下で、片方0より大きいなら大きい方から1減算して送信する
   Iterator((edge.srcId, edge.dstAttr - 1))
  }
 }else{
  if(edge.dstAttr <= 0){
   // 片方0以下で、片方0より大きいなら大きい方から1減算して送信する
   Iterator((edge.dstId, edge.srcAttr - 1))
  }else{
   // 両方とも0より大きいなら、双方の値を1減算して、双方に送信する
   val toSrc = Iterator((edge.srcId, edge.dstAttr - 1))
   val toDst = Iterator((edge.dstId, edge.srcAttr - 1))
   toDst ++ toSrc
  }
 }
}
```

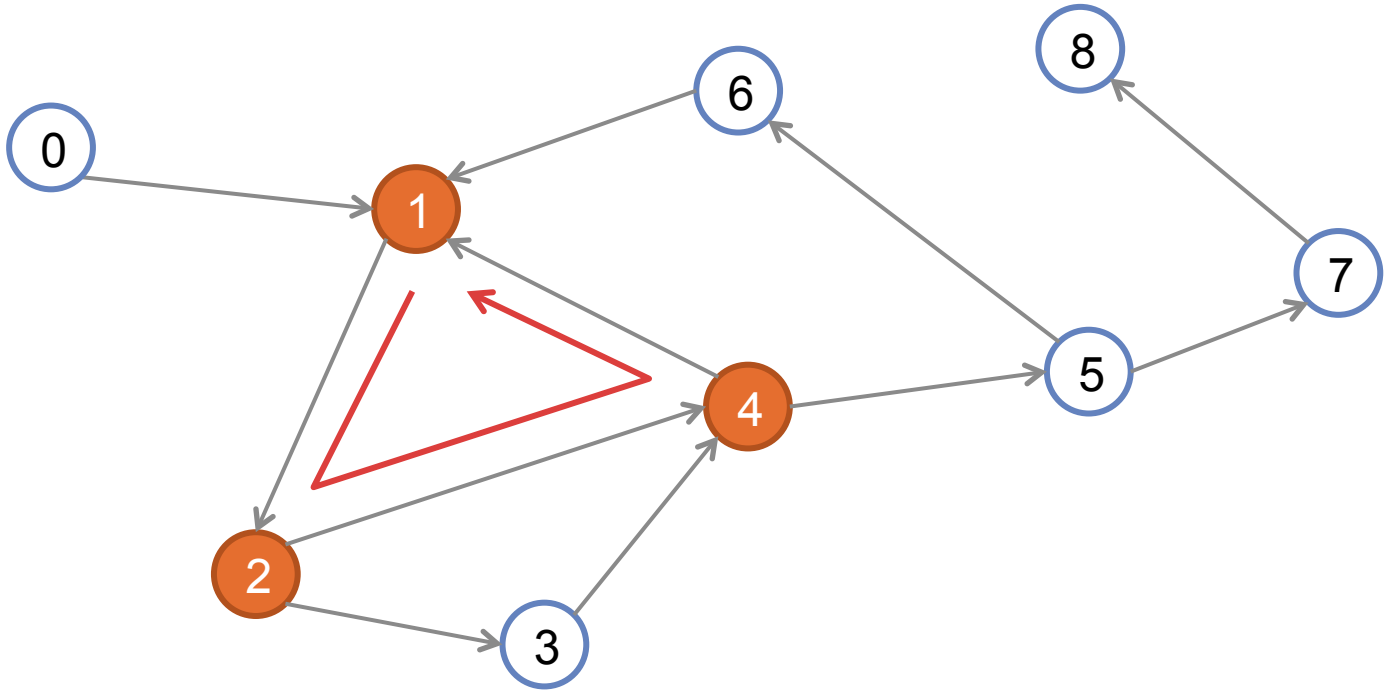
6．GraphX の Pregel 処理
②　輪の抽出

【課題】4ステップ以内の輪を探す



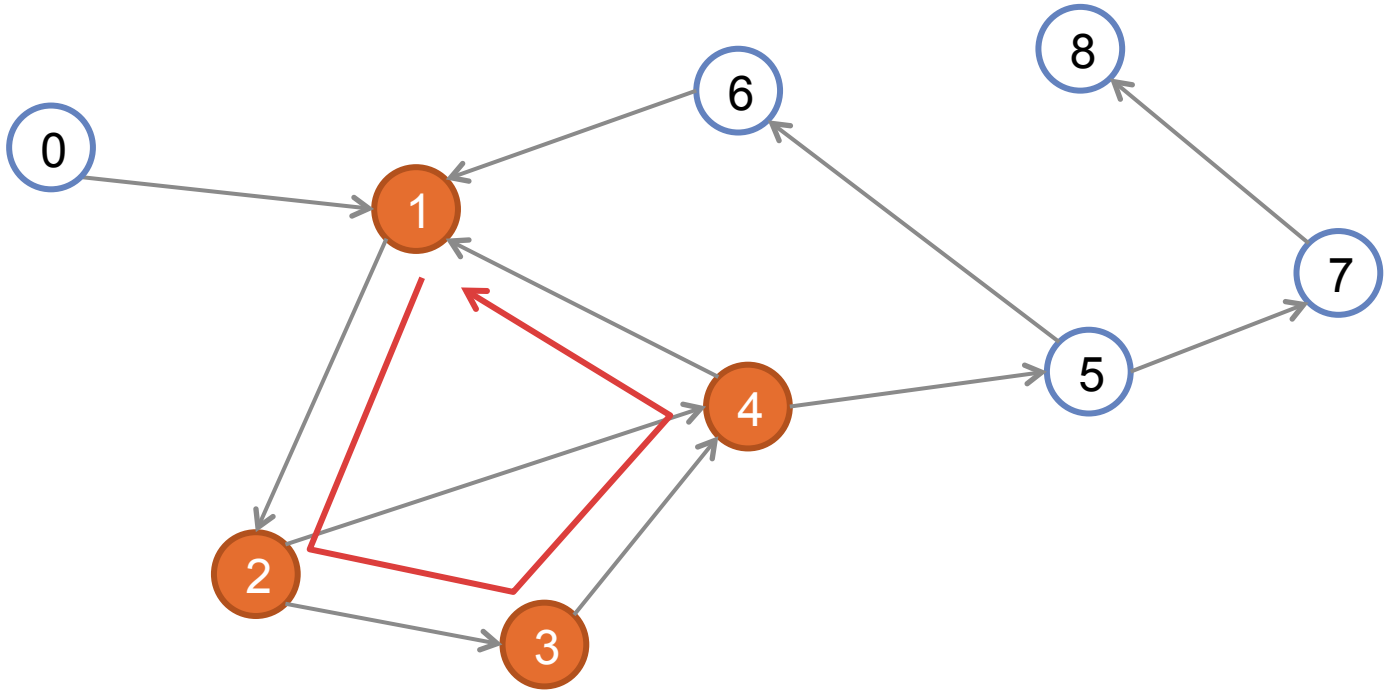
6．GraphX の Pregel 処理
②　輪の抽出

3ステップの輪　→　抽出対象



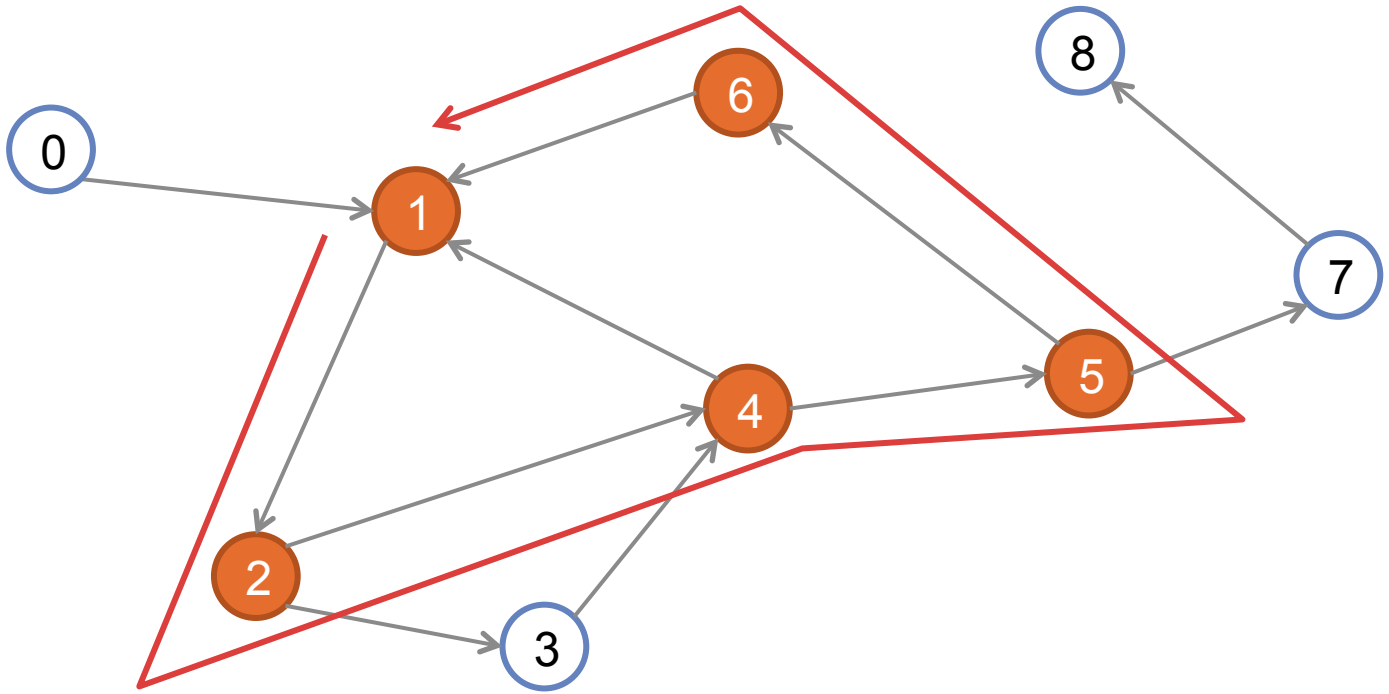
6．GraphX の Pregel 処理
②　輪の抽出

4ステップの輪　→　抽出対象



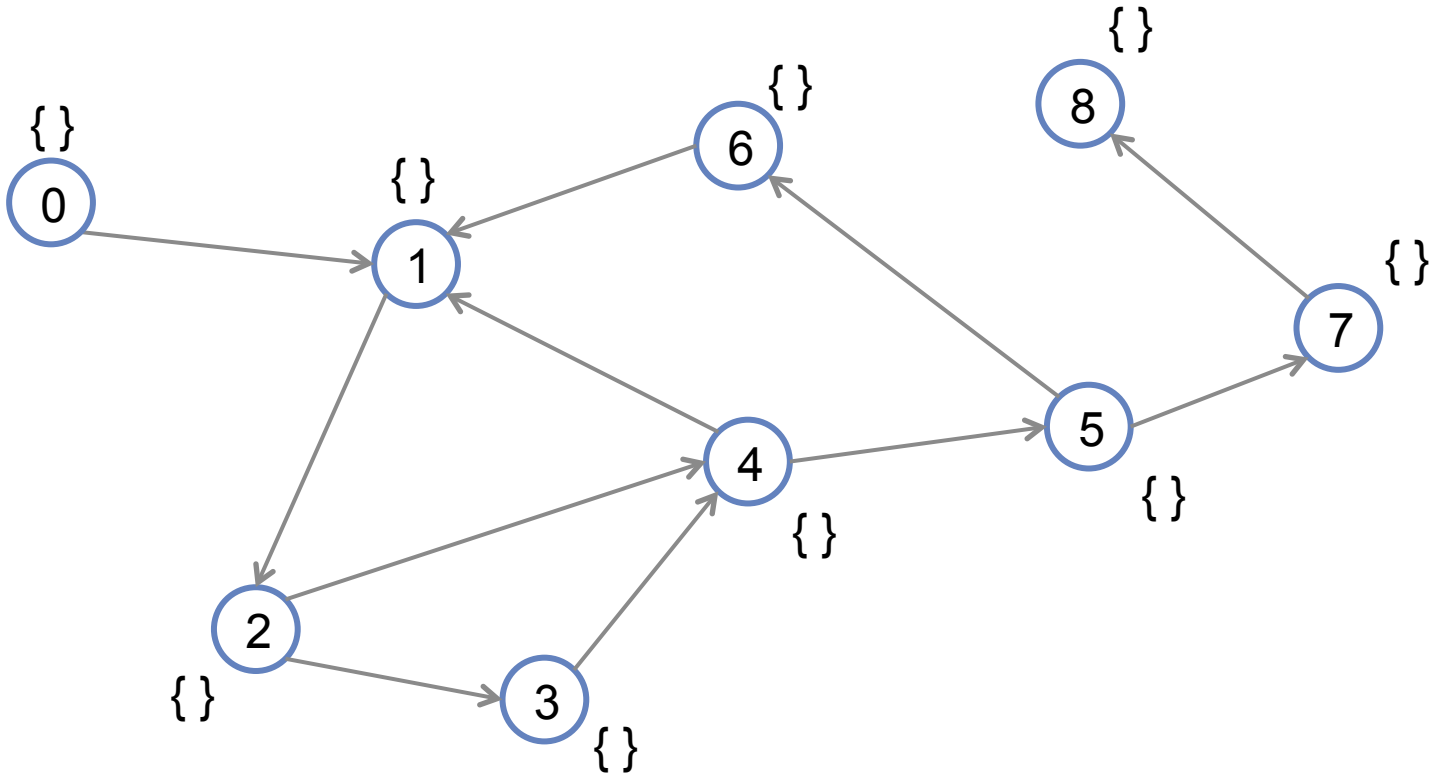
6．GraphX の Pregel 処理
②　輪の抽出

5ステップの輪　→　抽出対象外



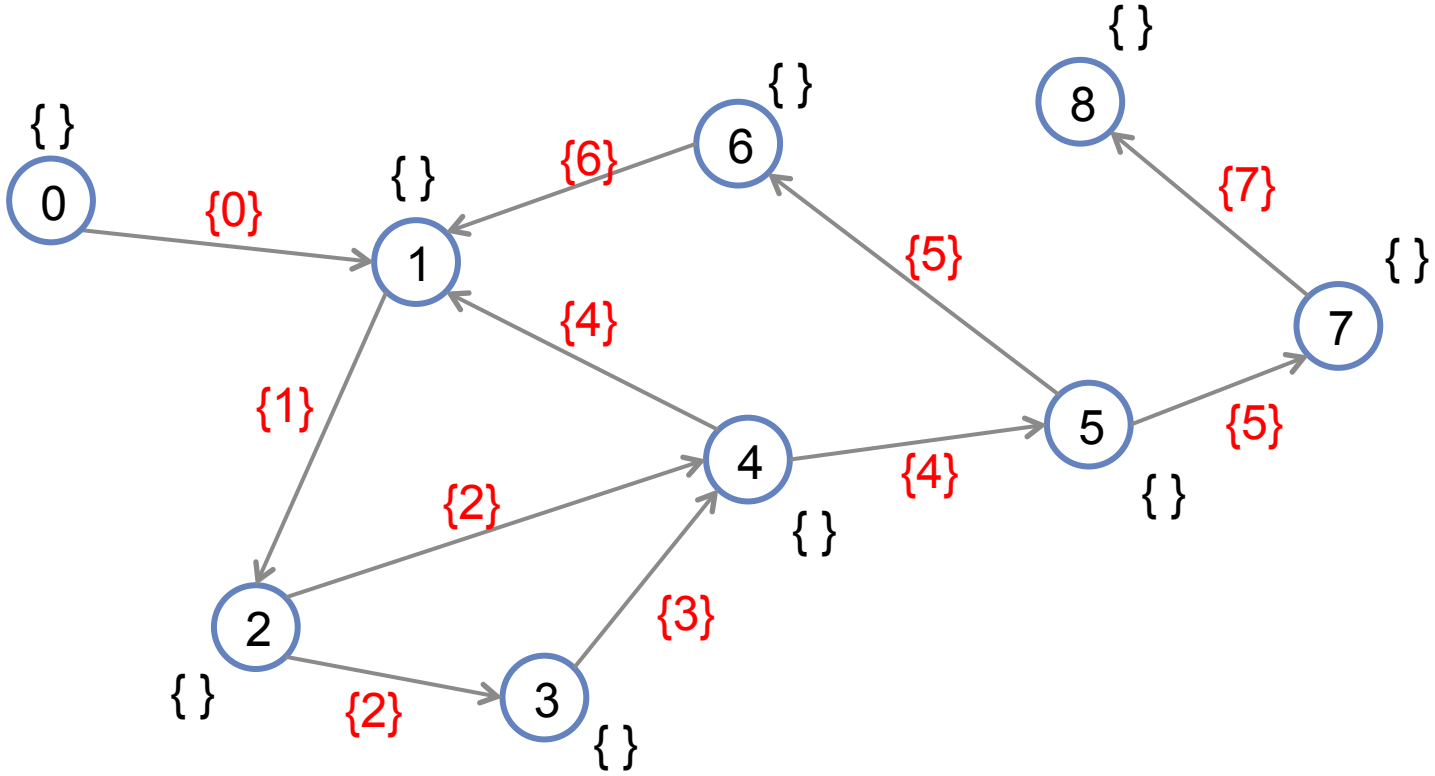
6．GraphX の Pregel 処理
②　輪の抽出

【事前処理】各頂点に空のリストを持たせる



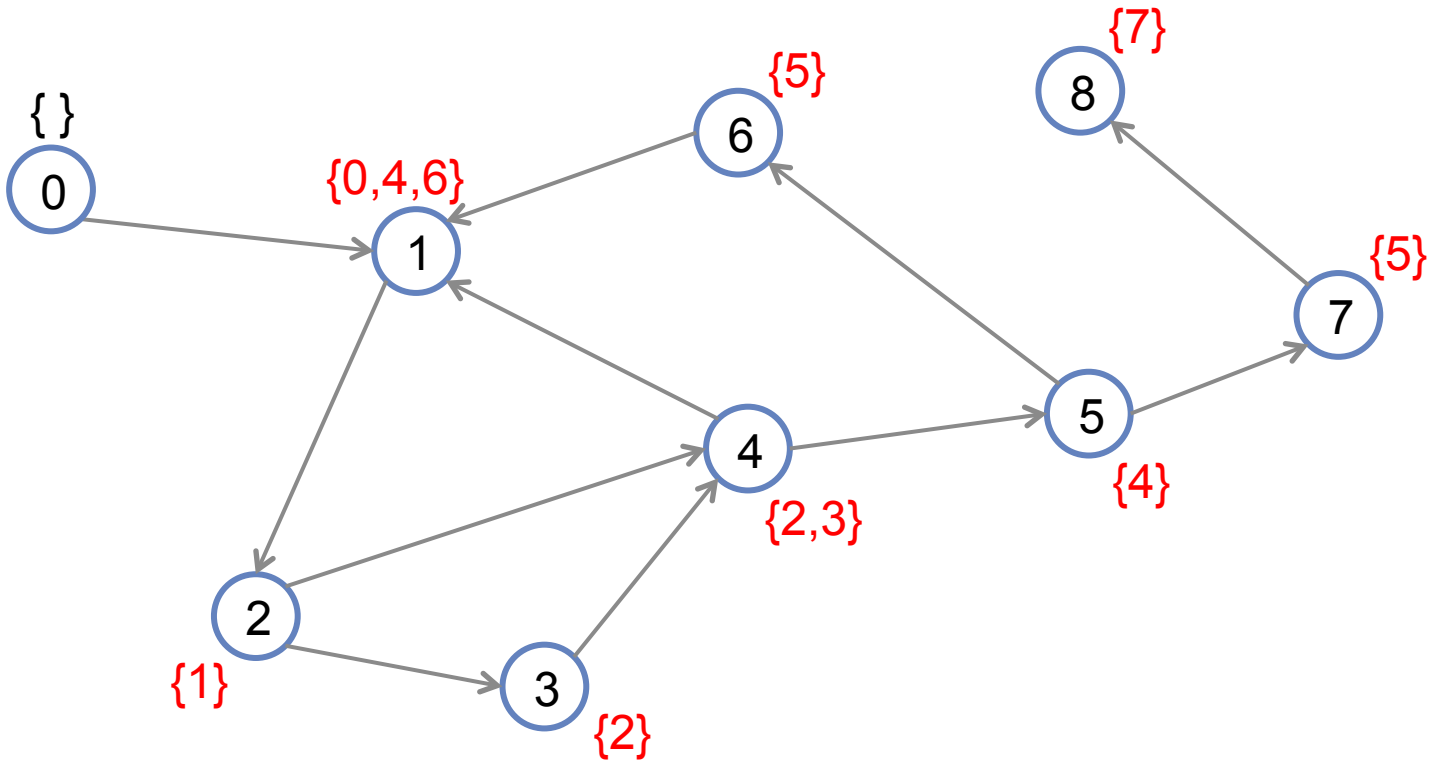
6．GraphX の Pregel 処理
②　輪の抽出

【Iteration1】自IDを自分のリストに格納して、接続先に送る



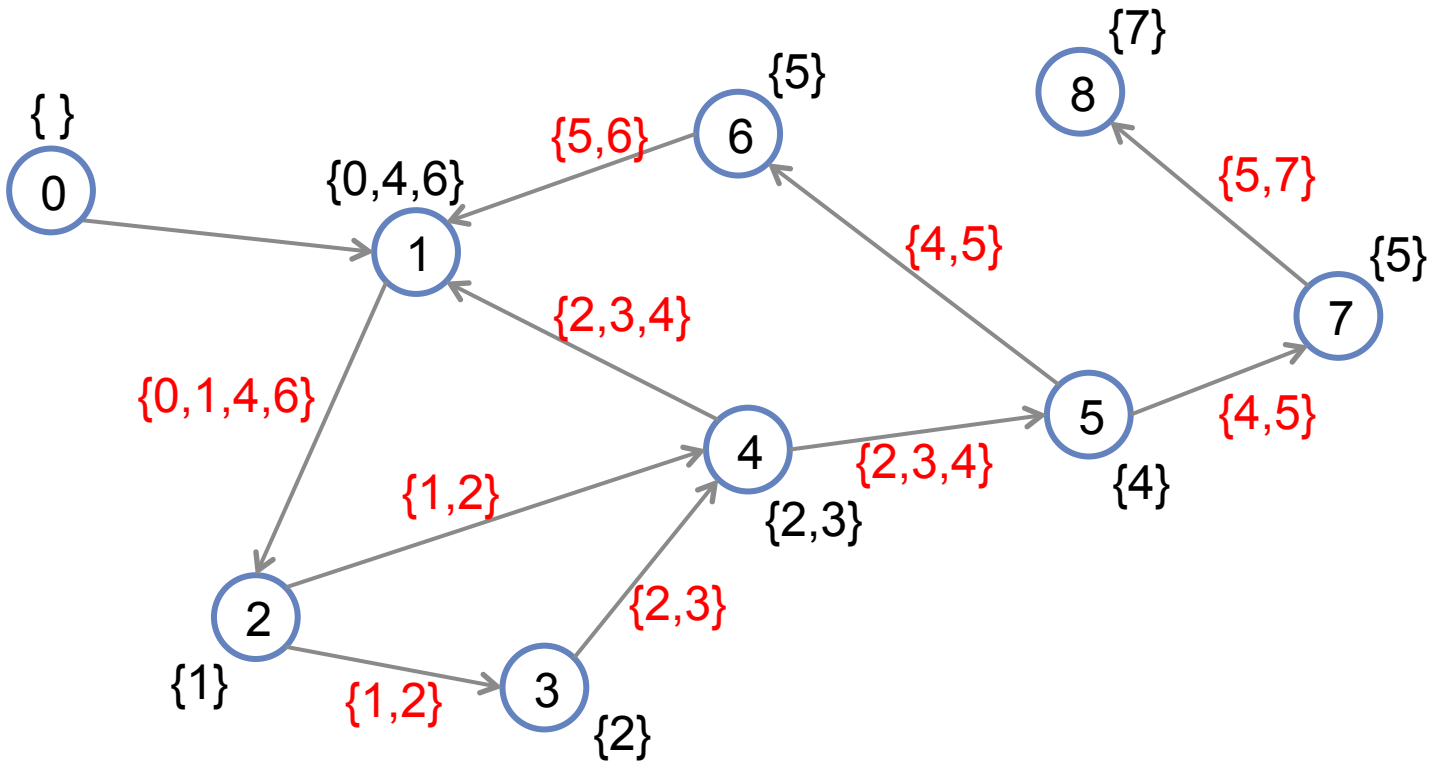
6．GraphX の Pregel 処理
②　輪の抽出

【Iteration1】受信したリストを自分のリストを合体させる



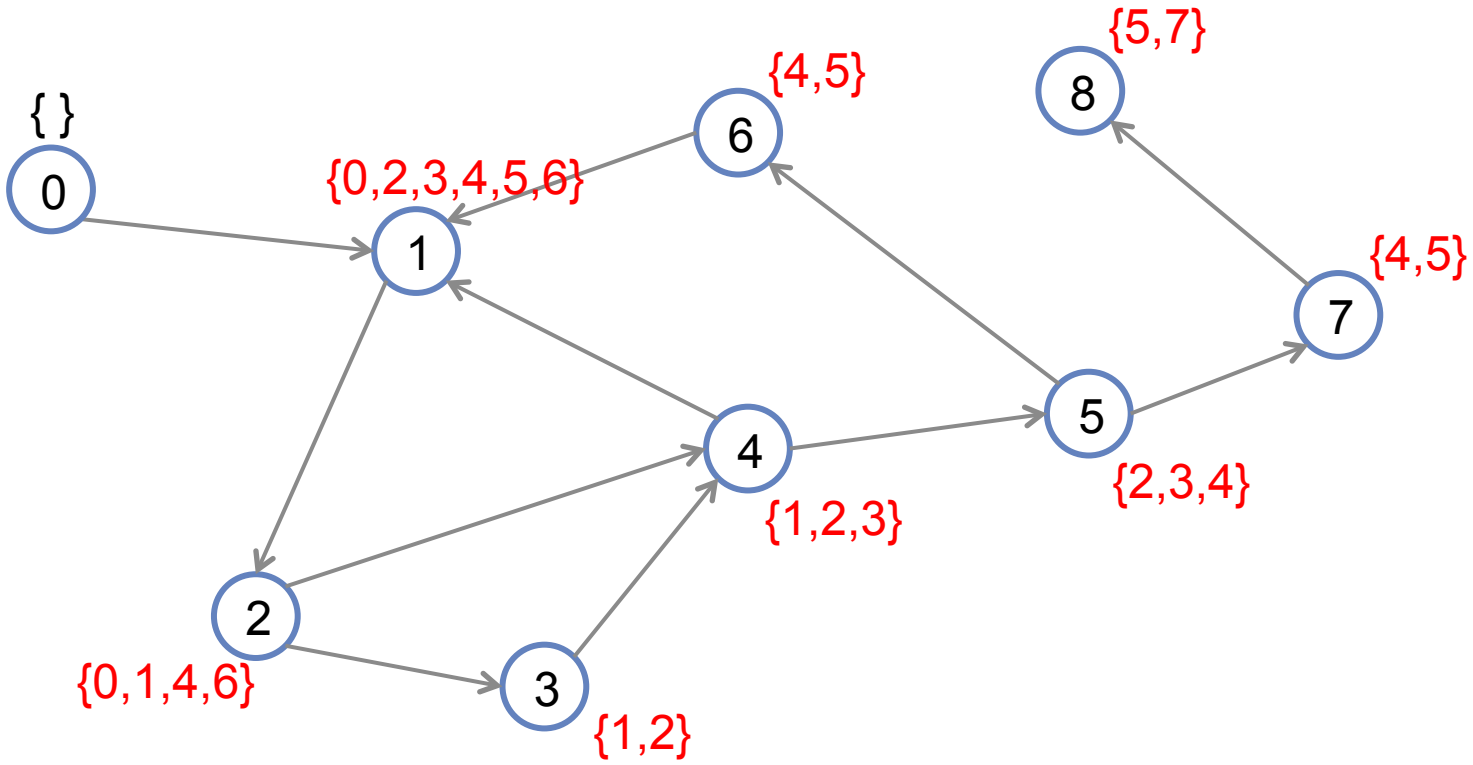
6．GraphX の Pregel 処理
②　輪の抽出

【Iteration2】自IDを自分のリストに格納して、接続先に送る



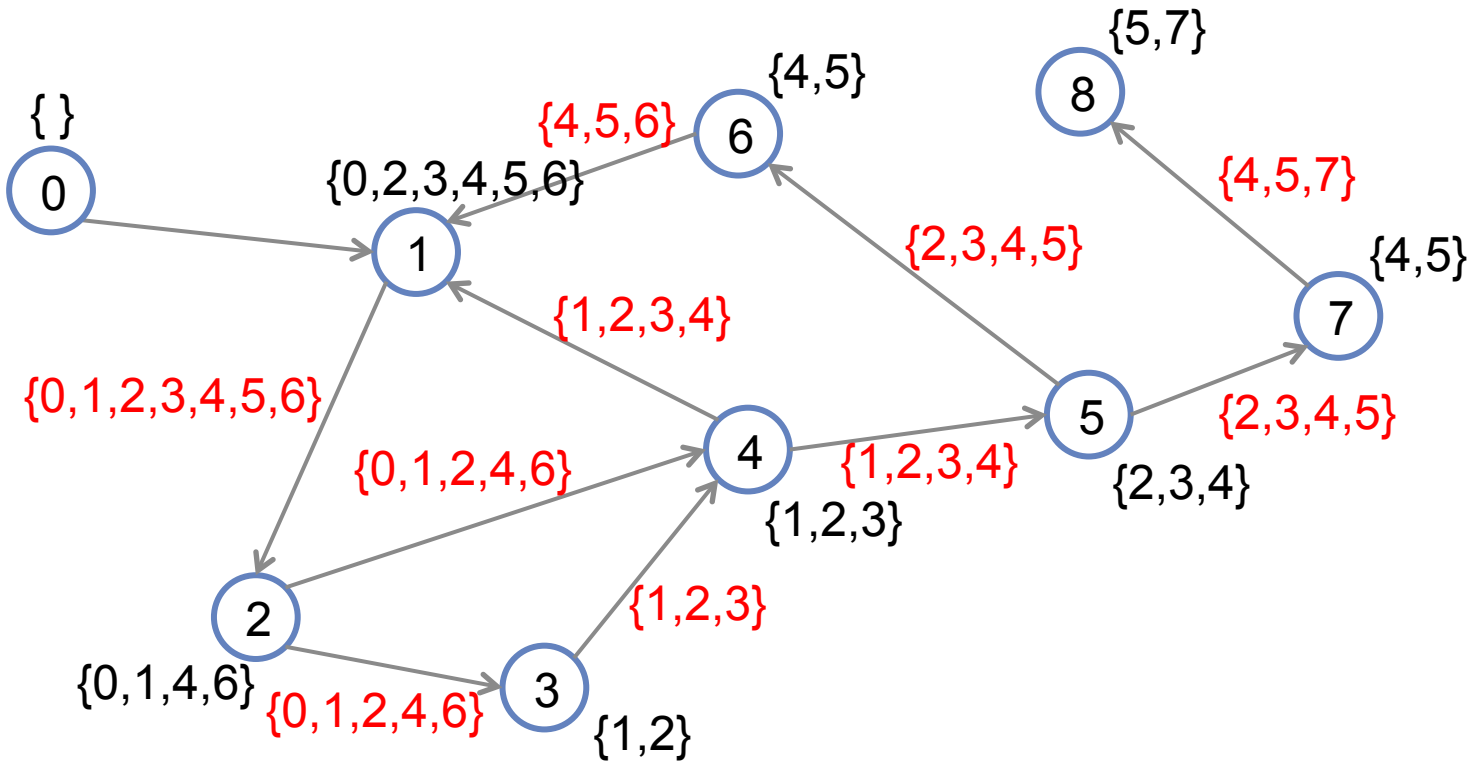
6．GraphX の Pregel 処理
②　輪の抽出

【Iteration2】受信したリストを自分のリストを合体させる



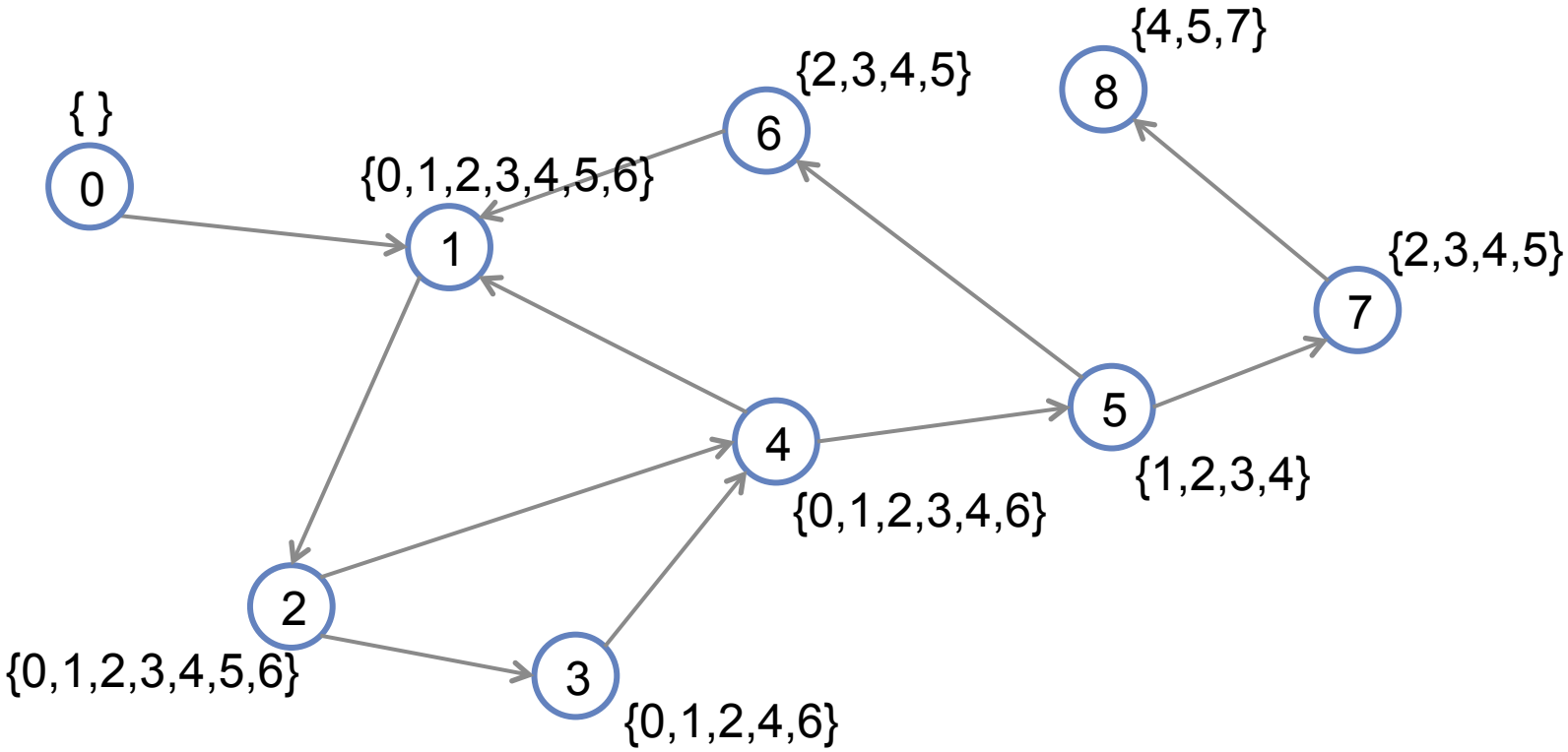
6．GraphX の Pregel 処理
②　輪の抽出

【Iteration3】自IDを自分のリストに格納して、接続先に送る



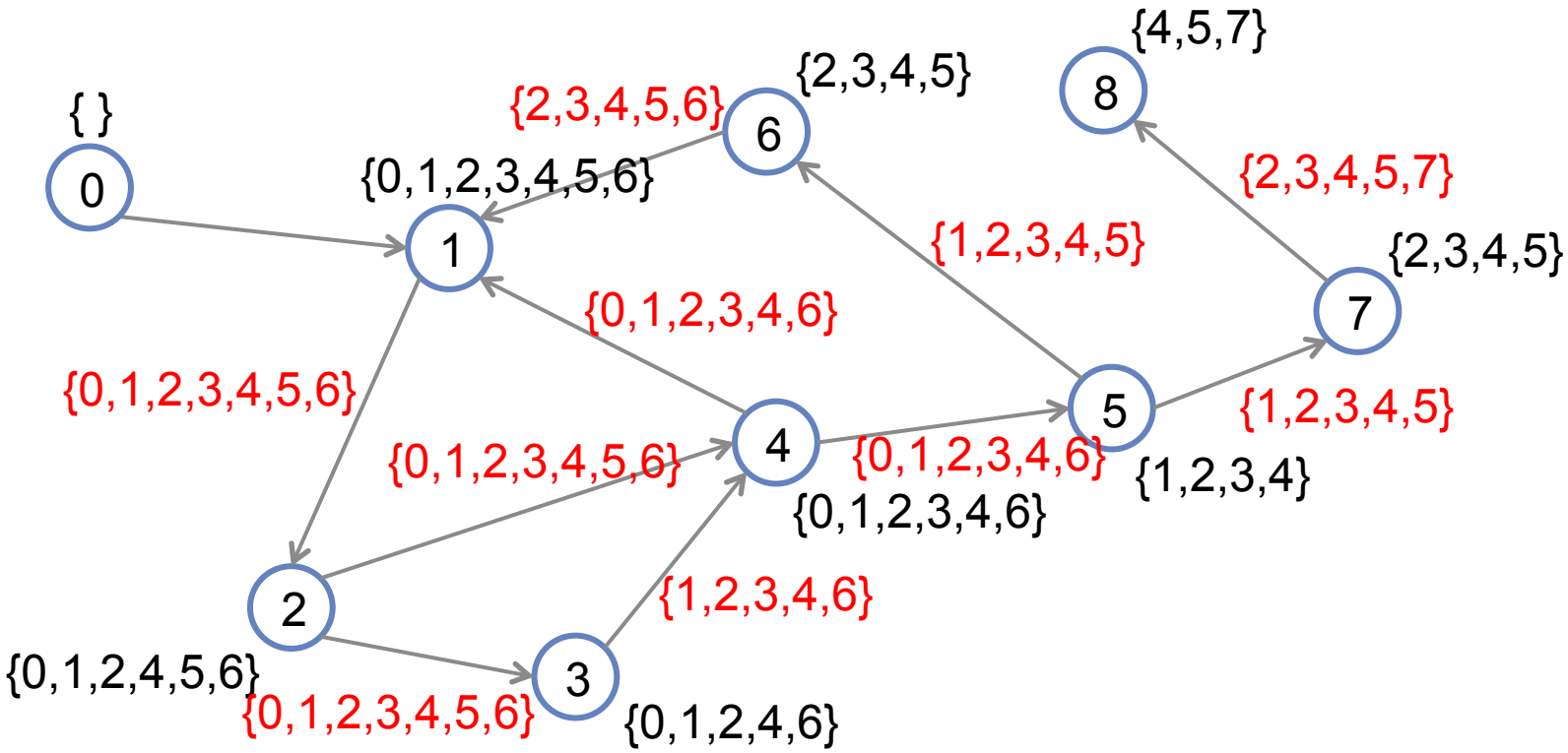
6．GraphX の Pregel 処理
②　輪の抽出

【Iteration3】受信したリストを自分のリストを合体させる



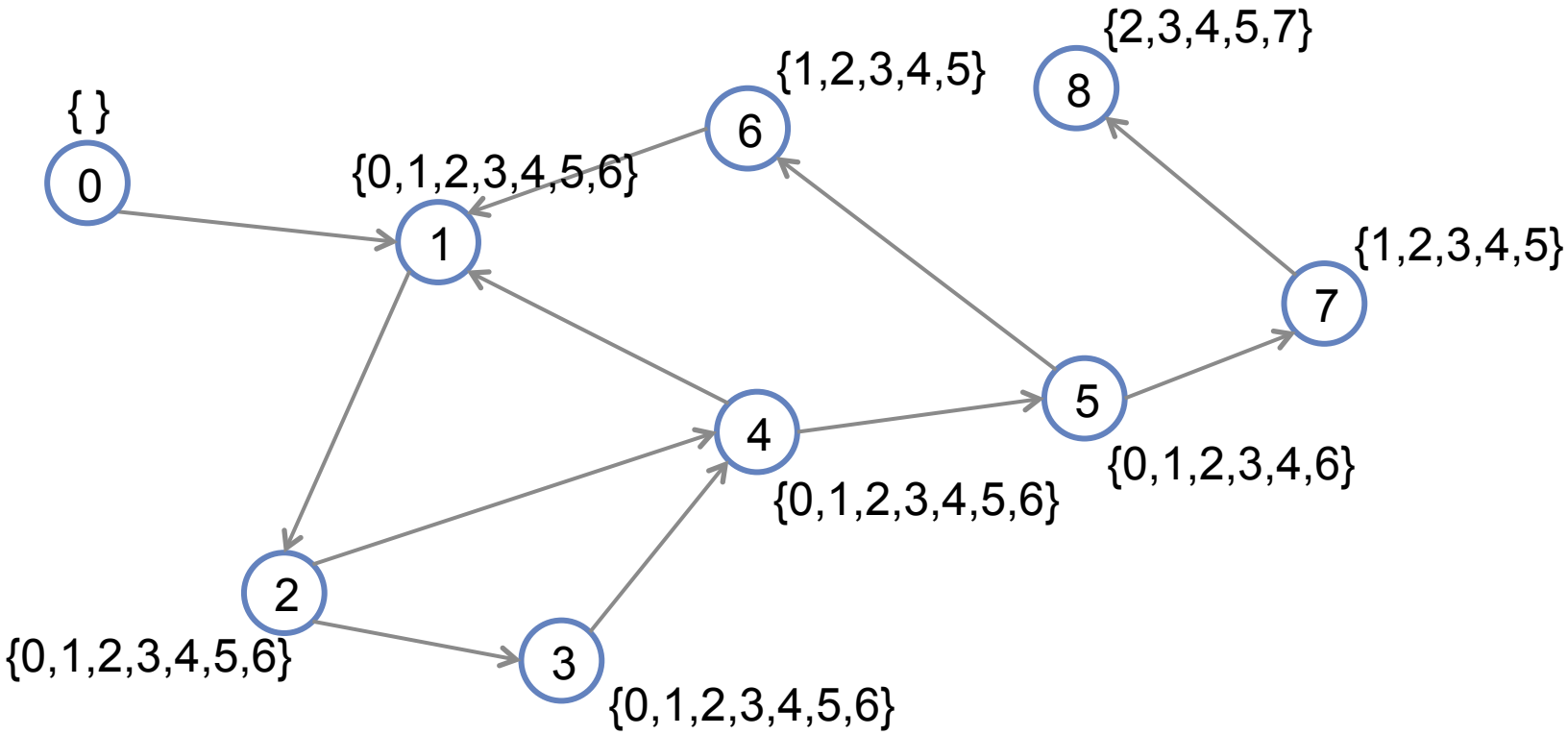
6．GraphX の Pregel 処理
②　輪の抽出

【Iteration4】自IDを自分のリストに格納して、接続先に送る



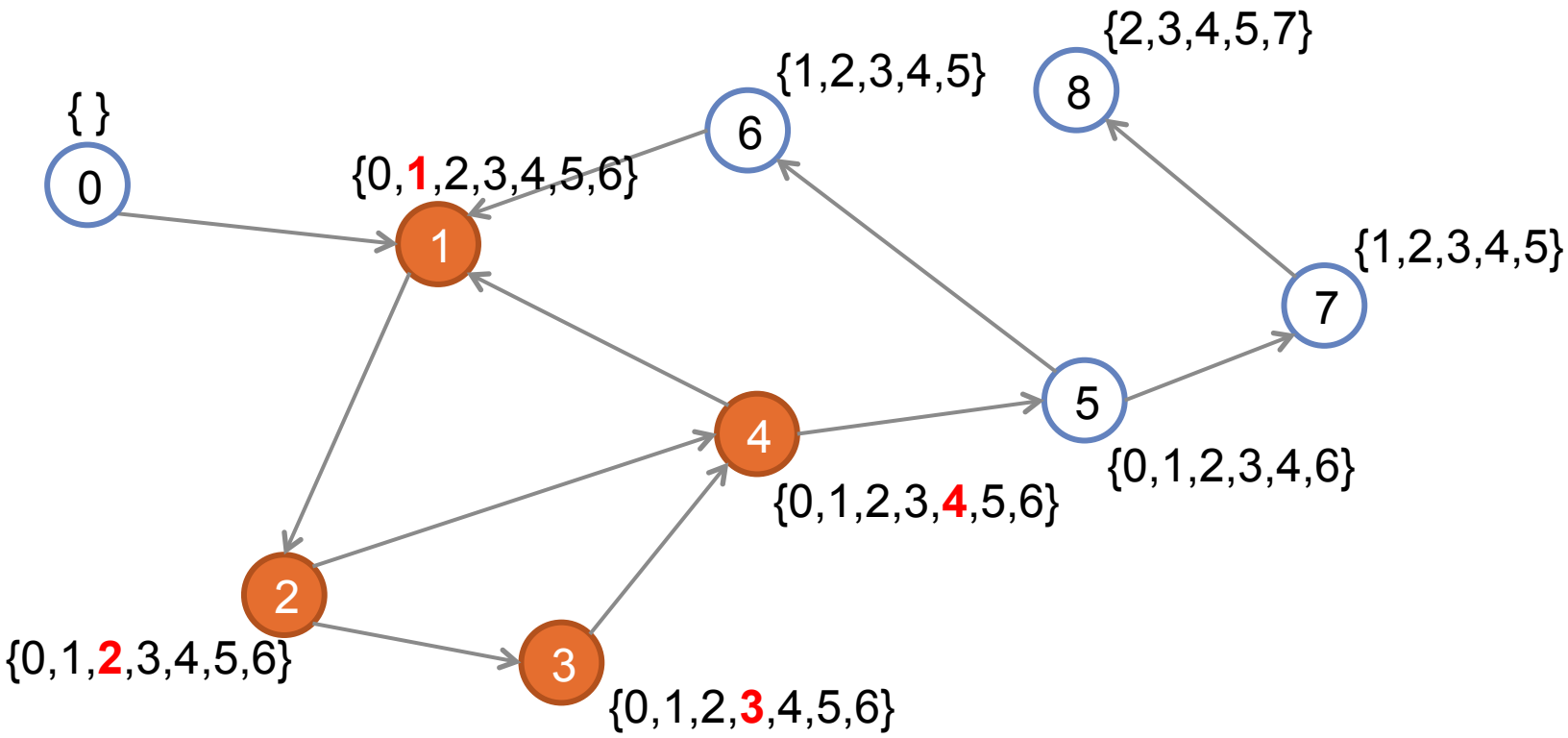
6．GraphX の Pregel 処理
②　輪の抽出

【Iteration4】受信したリストを自分のリストを合体させる



6．GraphX の Pregel 処理
②　輪の抽出

【Pregel後】自分のリストに、自分のIDが入っている頂点を抽出

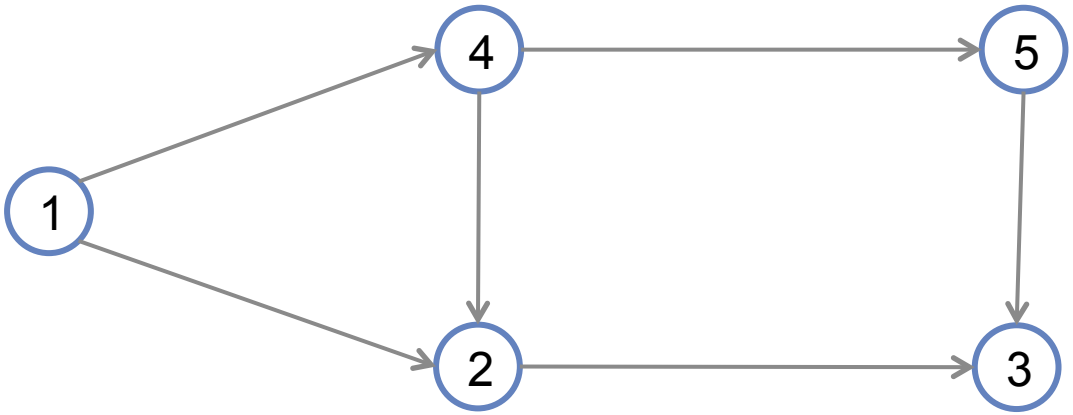


6．GraphX の Pregel 処理 ②　輪の抽出

```
val circleGraph = Pregel(
 // 各頂点に空のリストをセットしたグラフを処理する
graph.mapVertices((id, attr) => Set[VertexId]()),
 // 最初に辺を流すメッセージは空のグラフ
Set[VertexId](),
 // 4階層以内で還流しているところを探す
4,
 // メッセージを送る方向
EdgeDirection.Out) (
  // 自分のリストと、渡って来たリストを合体させる
 (id, attr, msg) => (msg ++ attr),
  // Srcが持っているリストにSrcのIDを追加してDstに渡す
 edge => Iterator((edge.dstId, (edge.srcAttr + edge.srcId))),
  // 複数Srcから送られて来たリストを合体させる
 (a, b) => (a ++ b)
// リストに自分のIDが入っている頂点が「輪」の中にいる頂点
).subgraph(vpred = (id, attr) => attr.contains(id))
```

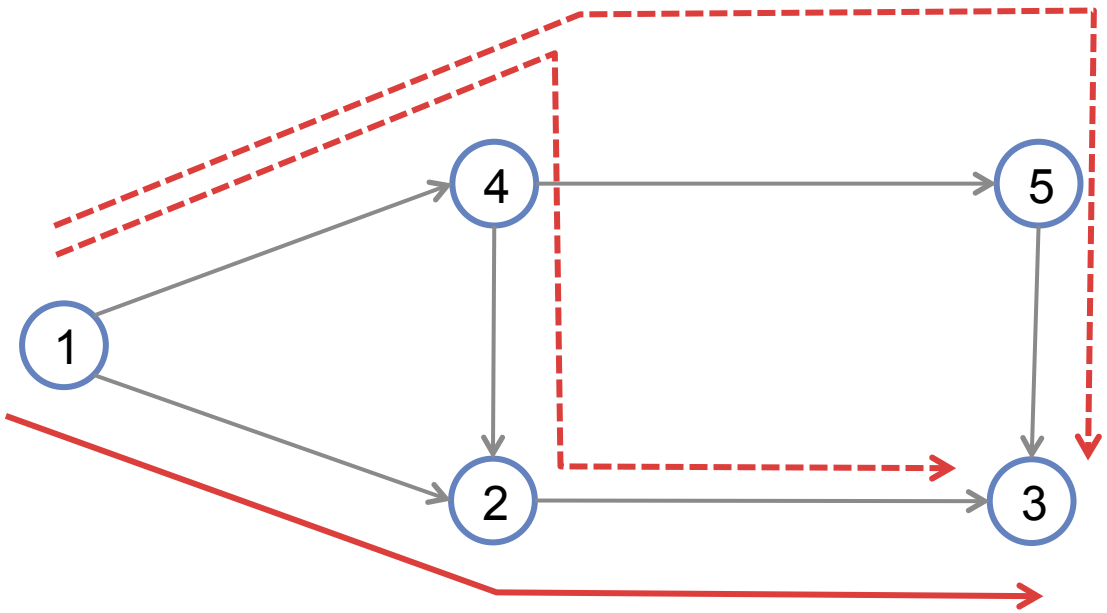
6．GraphX の Pregel 処理
③　距離の計測

【課題】各頂点間の最短距離を計測する



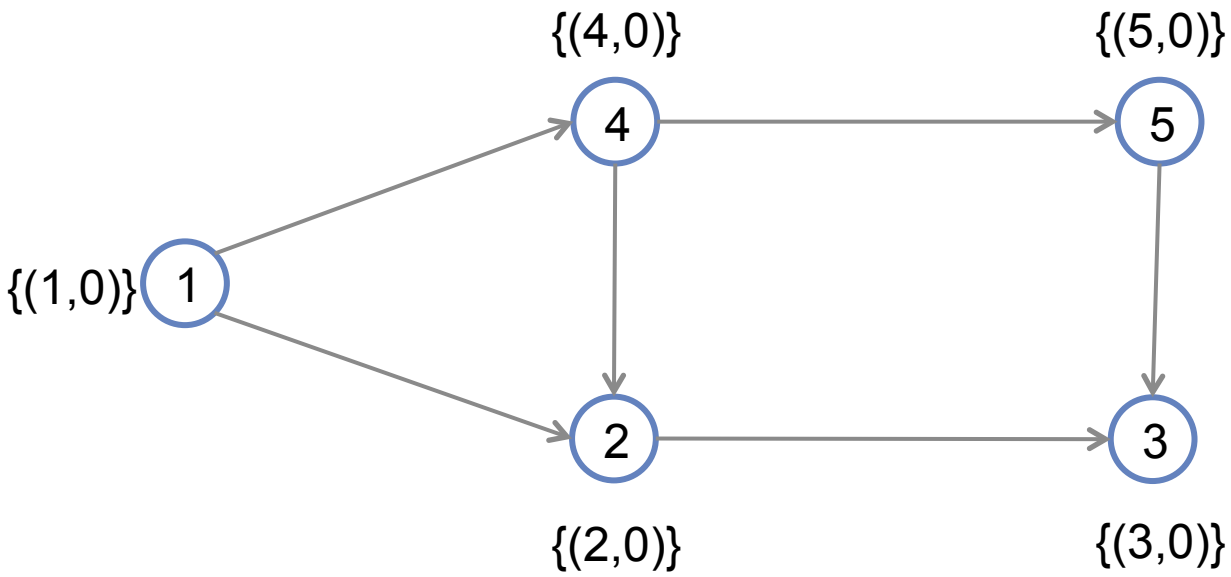
6．GraphX の Pregel 処理
③　距離の計測

【課題】各頂点間の最短距離を計測する



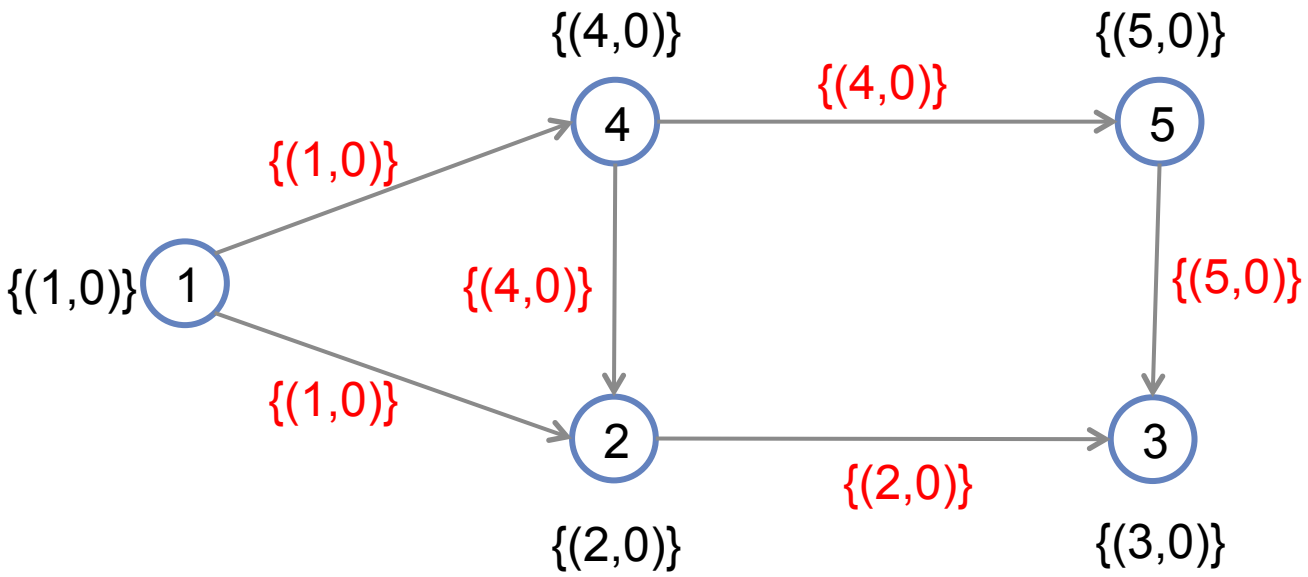
6．GraphX の Pregel 処理
③　距離の計測

【事前処理】各頂点にリストを持たせる。リストの1行目として、自分自身と、その距離（0）を持つ



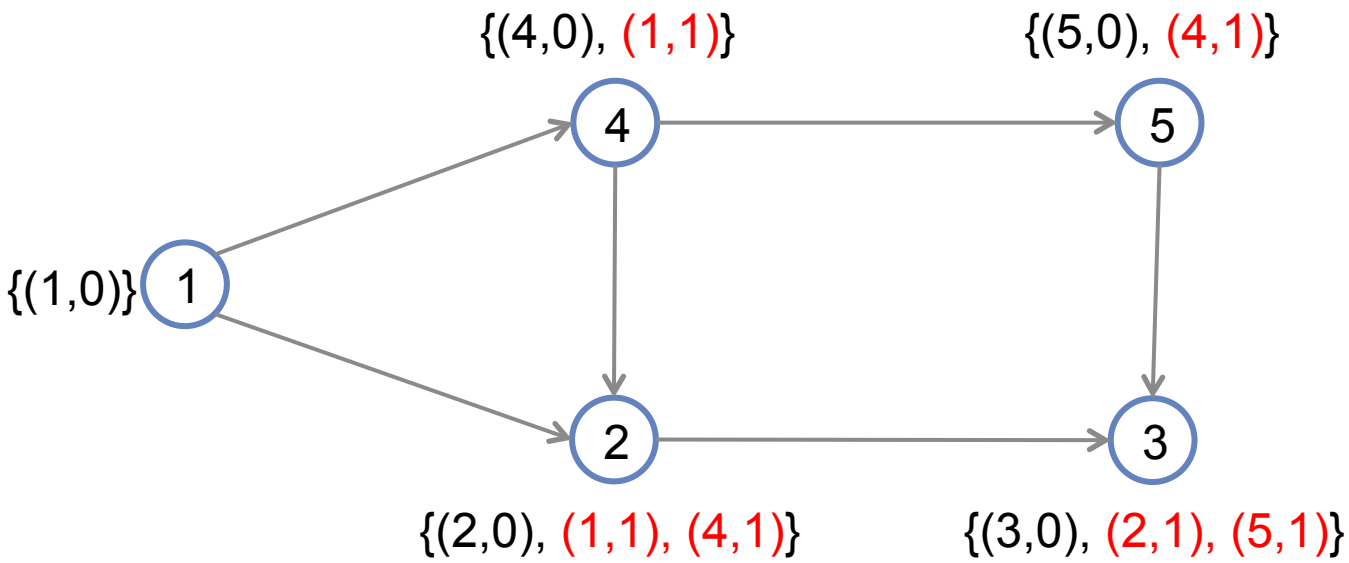
6．GraphX の Pregel 処理
③　距離の計測

【Iteration1】自分が持つリストを、接続先頂点に送る



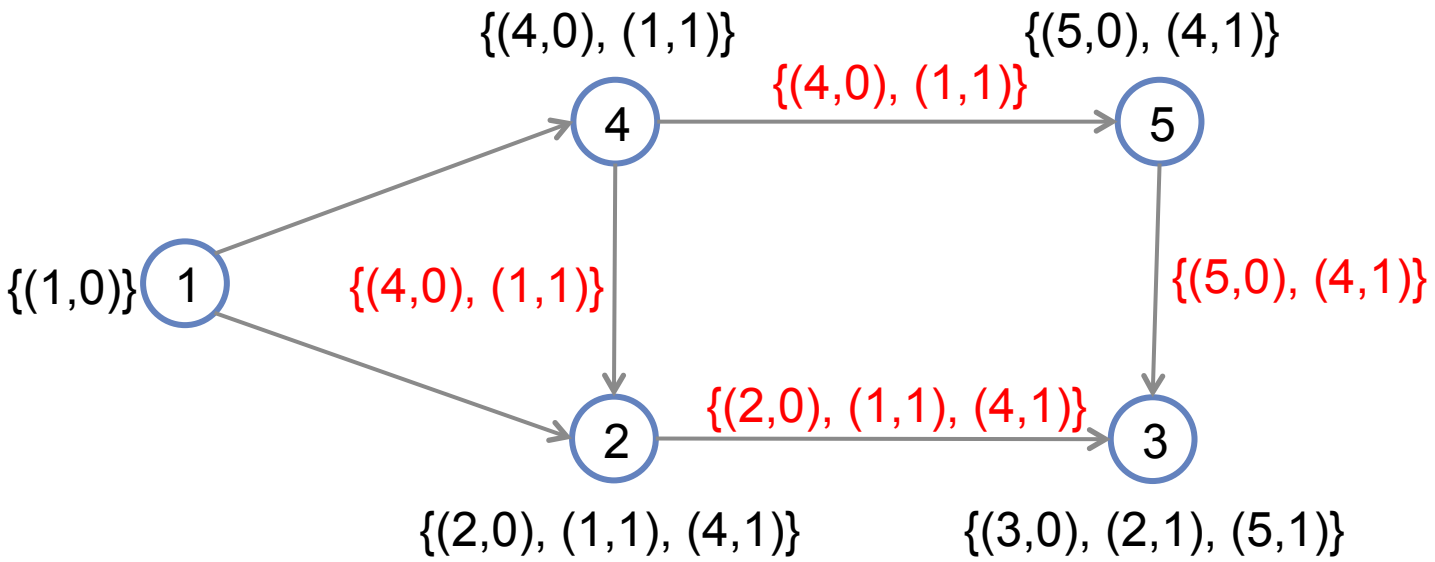
6．GraphX の Pregel 処理
③　距離の計測

【Iteration1】受信したリストを、距離を1加算しつつ、自分のリストにマージする



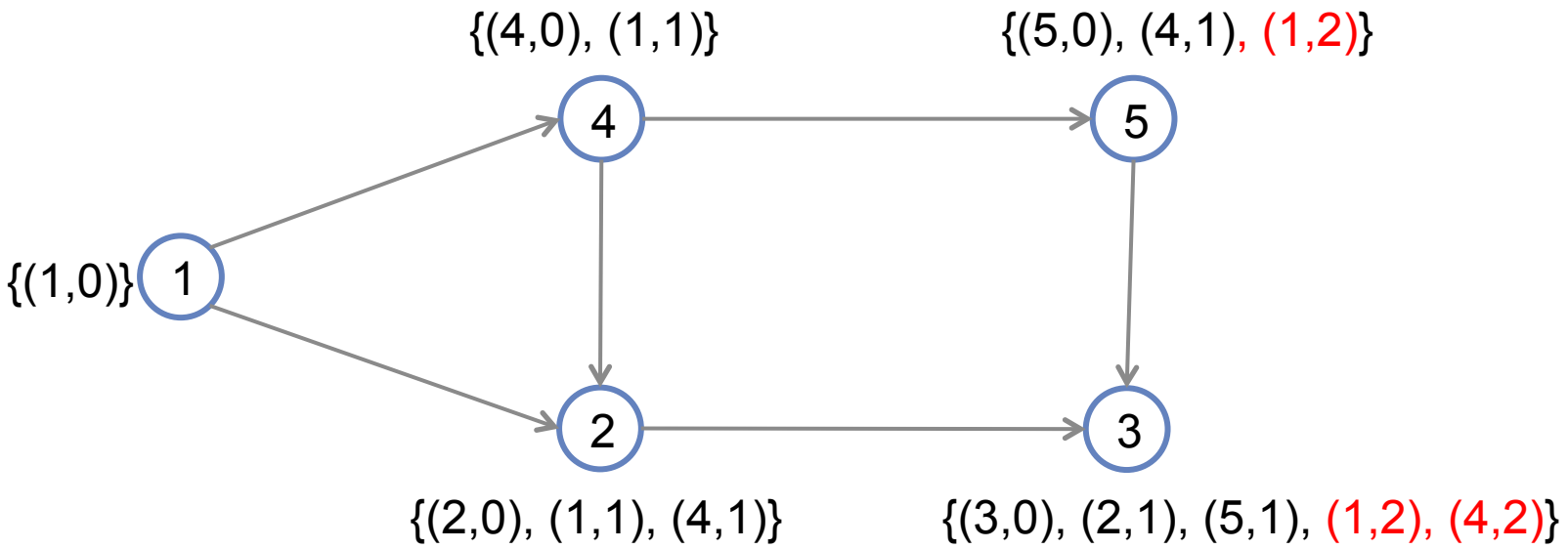
6．GraphX の Pregel 処理
③　距離の計測

【Iteration2】自分が持つリストを、接続先頂点に送る
（直前のIterationで受信した頂点だけが送信処理を稼動させる
（Pregelの仕様））



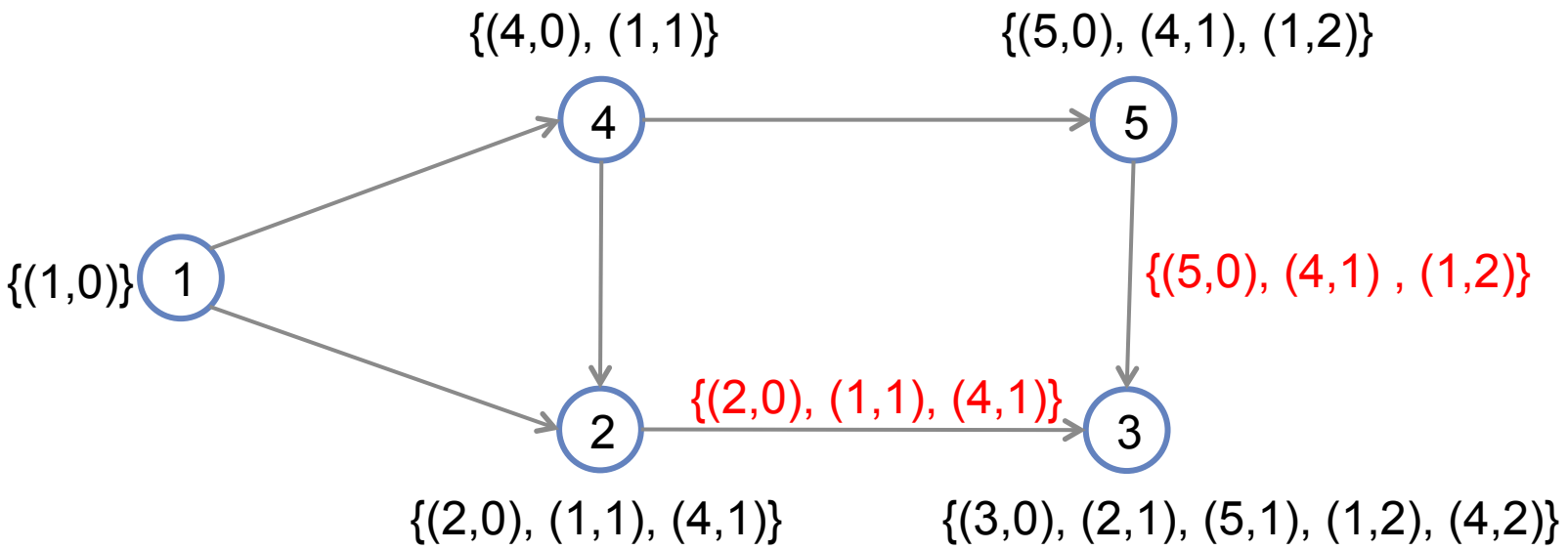
6．GraphX の Pregel 処理
③　距離の計測

【Iteration2】受信したリストを、距離を1加算しつつ、自分のリストにマージする。既にリストに頂点情報がある場合は、距離が短い方を選択する。



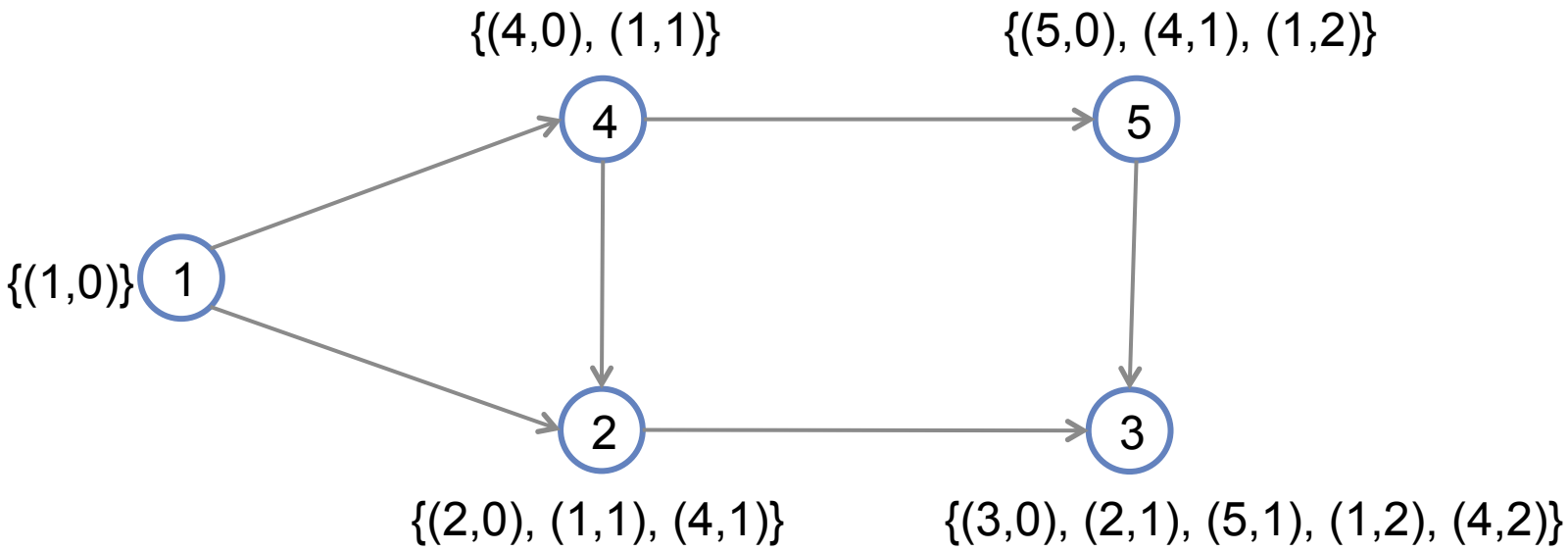
6．GraphX の Pregel 処理
③　距離の計測

【Iteration3】自分が持つリストを、接続先頂点に送る
（直前のIterationで受信した頂点だけが送信処理を稼動させる
（Pregelの仕様））

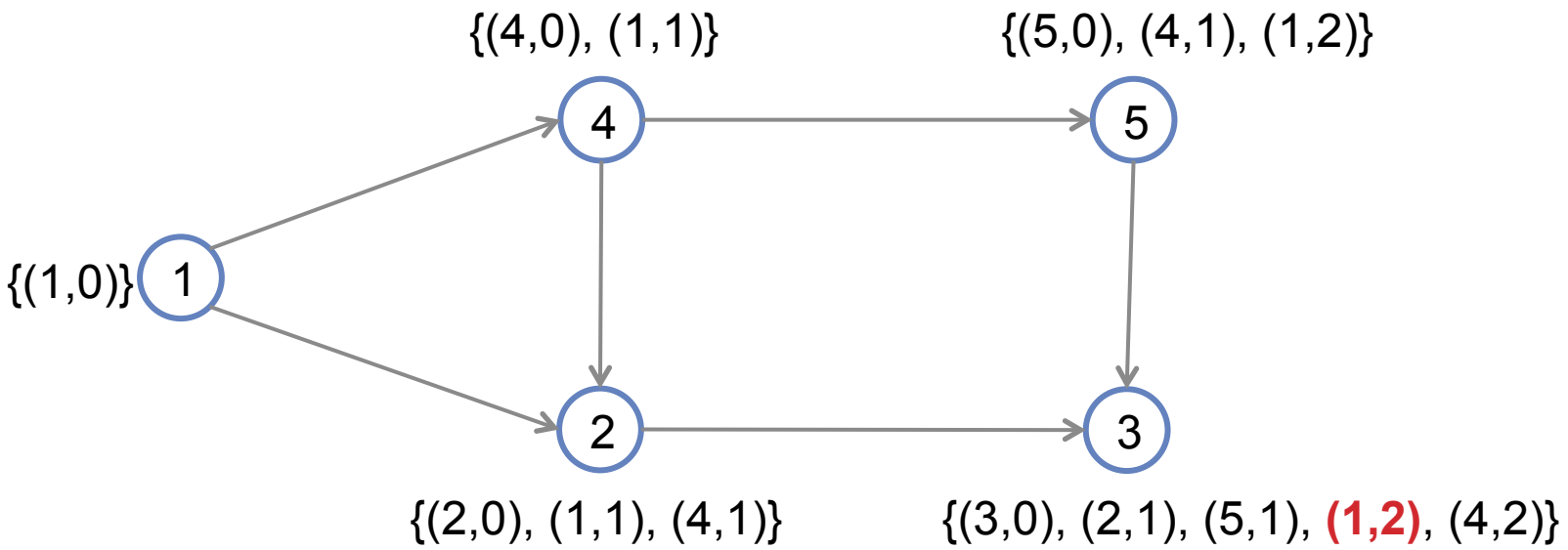


6．GraphX の Pregel 処理
③　距離の計測

【Iteration3】受信したリストを、距離を1加算しつつ、自分のリストにマージする。既にリストに頂点情報がある場合は、距離が短い方を選択する。



6．GraphX の Pregel 処理
③　距離の計測



6．GraphX の Pregel 処理
③　距離の計測

```
val graphWithDistance = Pregel(
  // 他頂点の距離のリストを頂点に持たせる
 graph.mapVertices((id:VertexId, attr:Int) => List((id, 0))),
  List[(VertexId, Int)](), //最初に流すメッセージは空のリスト
  Int.MaxValue, // イテレーション回数は指定しない
  EdgeDirection.Out // Out方向にメッセージ送信する
  )(
  // 各頂点では送られて来たメッセージ内にある他頂点とその距離に対して1を足して行き、
 // それと自分が持つ情報とを、距離が短い方を選択しつつ集約していく。
 (id, attr, msg) =>
    mergeVertexRoute(attr, msg.map(a=> (a._1, a._2 + 1))),
  edge => {
   // srcが持つ他頂点との距離リストの中に、dstの頂点IDがあれば循環していると判断
  val isCyclic = edge.srcAttr.filter(_._1 == edge.dstId).nonEmpty
   // 循環しているようであれば、既に距離は計測済みなので何も送信しない。
  // さもなければ、dstに対してsrcが持つ、他頂点とその距離のリストを送る
  if(isCyclic) Iterator.empty
     else Iterator((edge.dstId, edge.srcAttr))},
 // 複数のsrcから送られて来たリストは単純合体させる
 (m1, m2) => m1 ++ m2
)
```

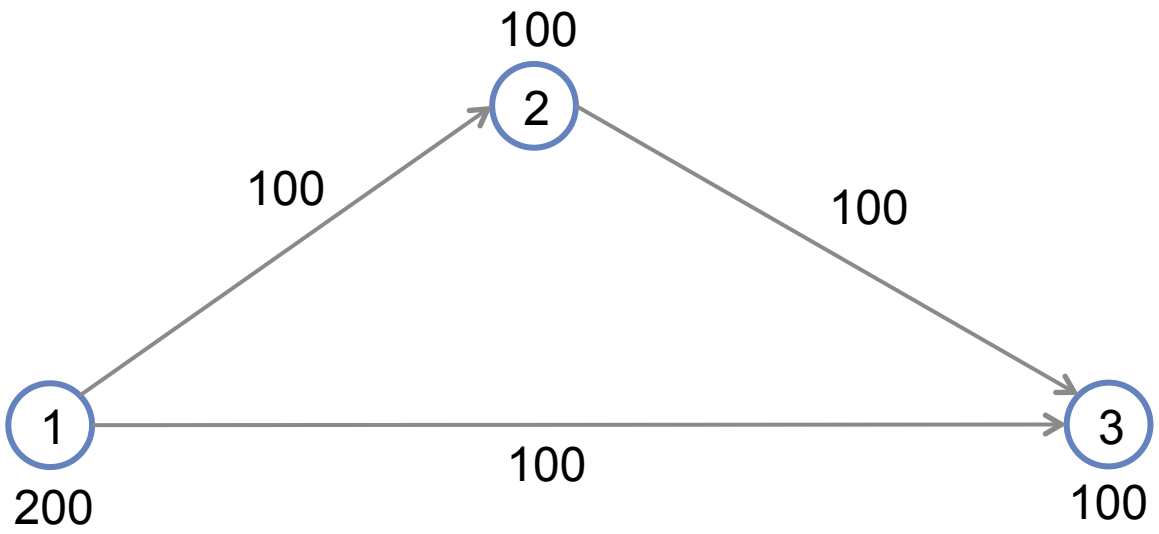
6．GraphX の Pregel 処理
③　距離の計測

```
// 短い方の距離を選定
def minRouteDepth(v1:(VertexId, Int), v2:(VertexId, Int))
 = if(v1._2 < v2._2) v1 else v2

// 頂点毎に持つ、他頂点との距離情報の集約
def mergeVertexRoute(oldRoute:List[(VertexId, Int)],
newRoute:List[(VertexId, Int)])
  = (oldRoute ++ newRoute)
    //頂点Idでグルーピング
    .groupBy(_._1)
    //頂点Idでグルーピングされた中で距離の短い方を選択
    .map(_._2.reduce((v1, v2) => minRouteDepth(v1, v2)))
    .toList
```

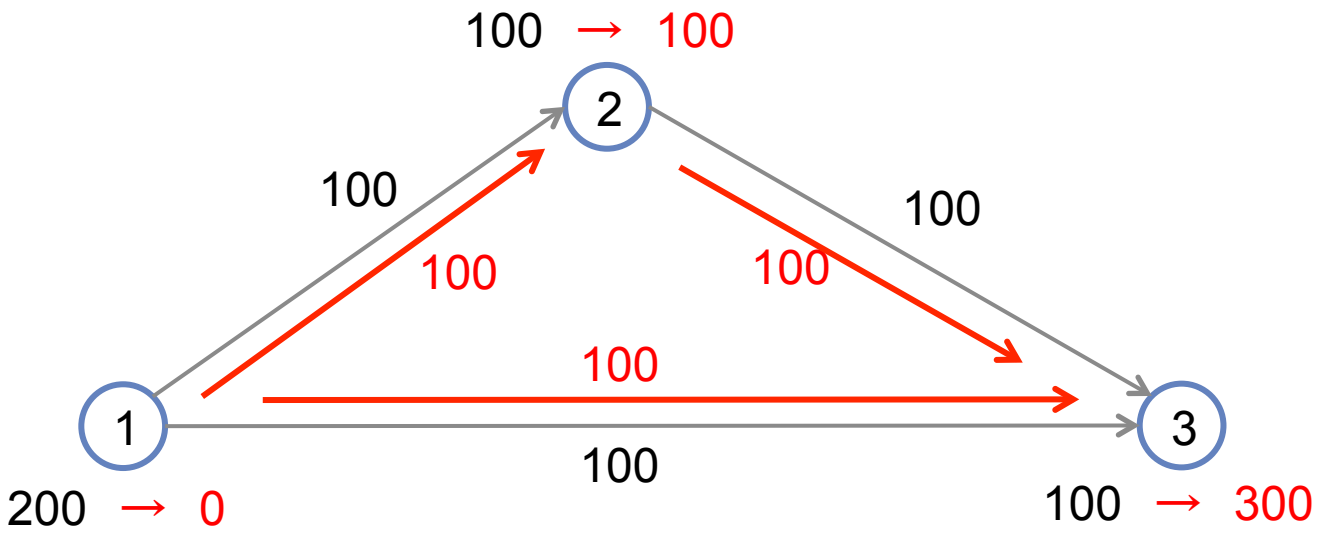
6．GraphX の Pregel 処理
④　影響の計測

【考え方】相互に以下のような関係性があるとする



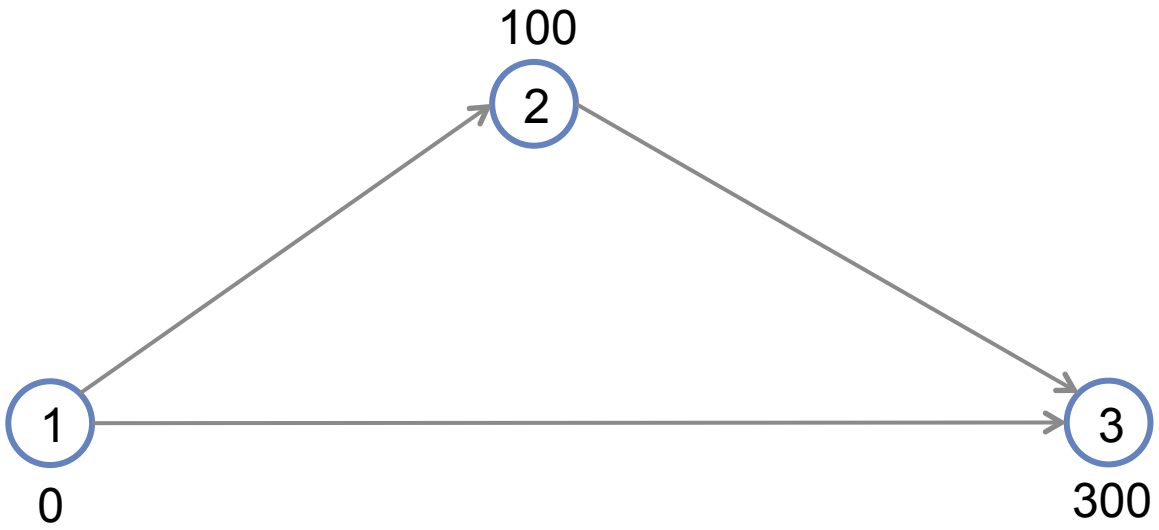
6．GraphX の Pregel 処理
④　影響の計測

【考え方】関係性の清算を行う



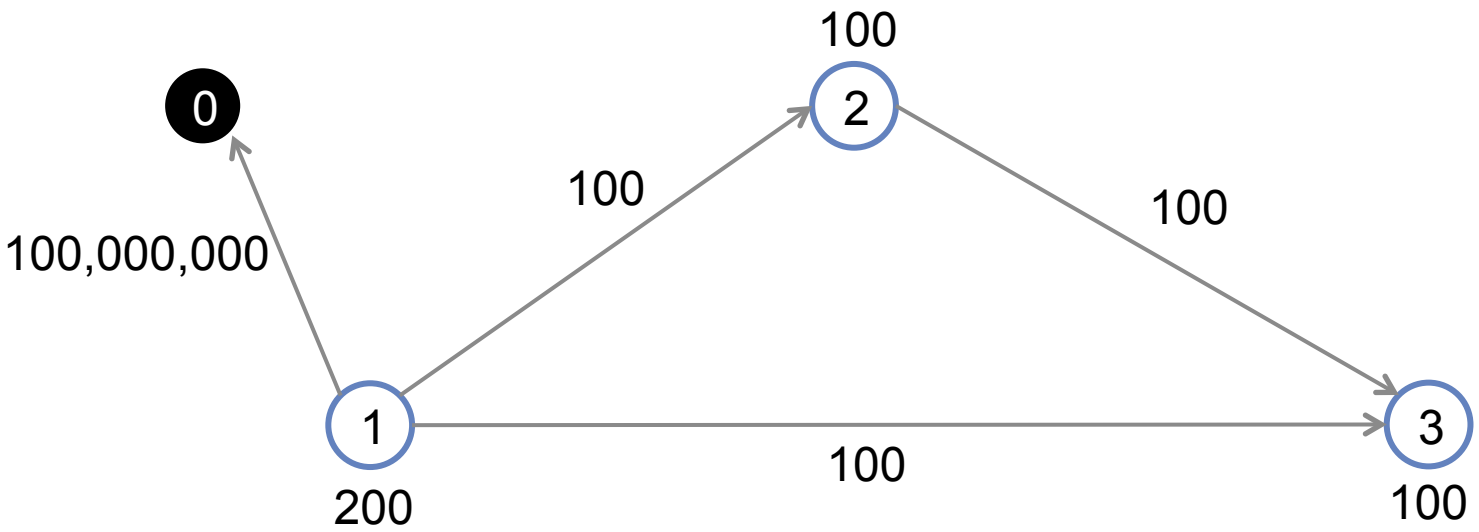
6．GraphX の Pregel 処理
④　影響の計測

【考え方】清算後の状態



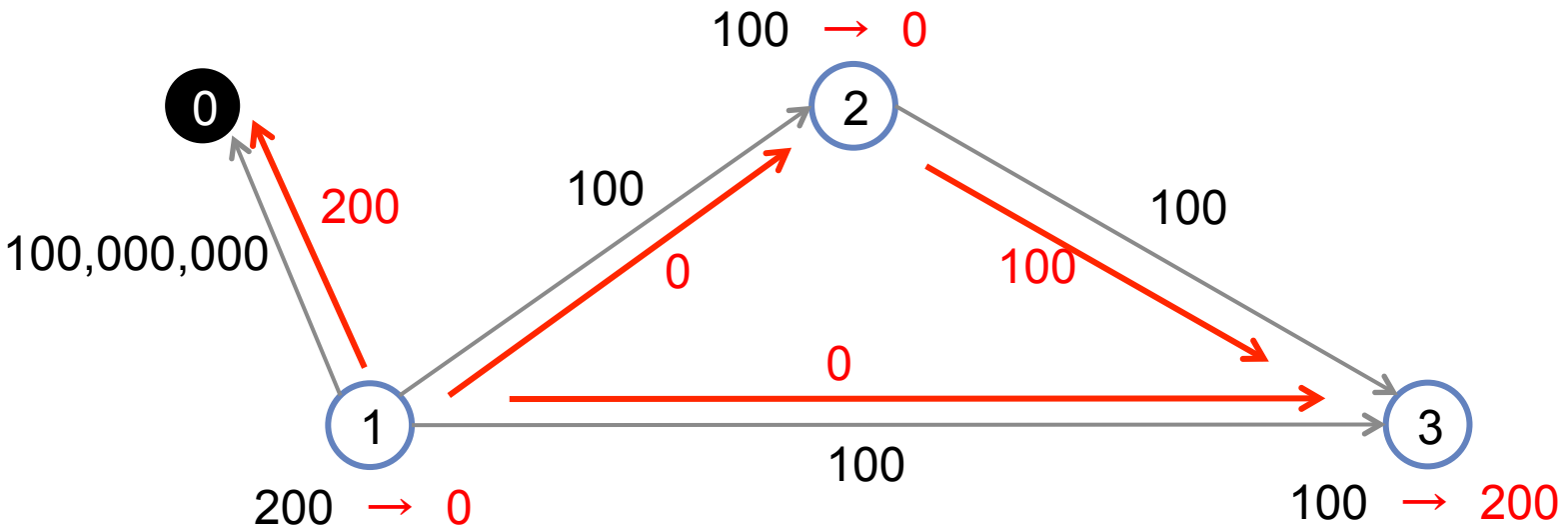
6．GraphX の Pregel 処理
④　影響の計測

【考え方】「１」の影響を知るために１の存在を消す関係性を加える



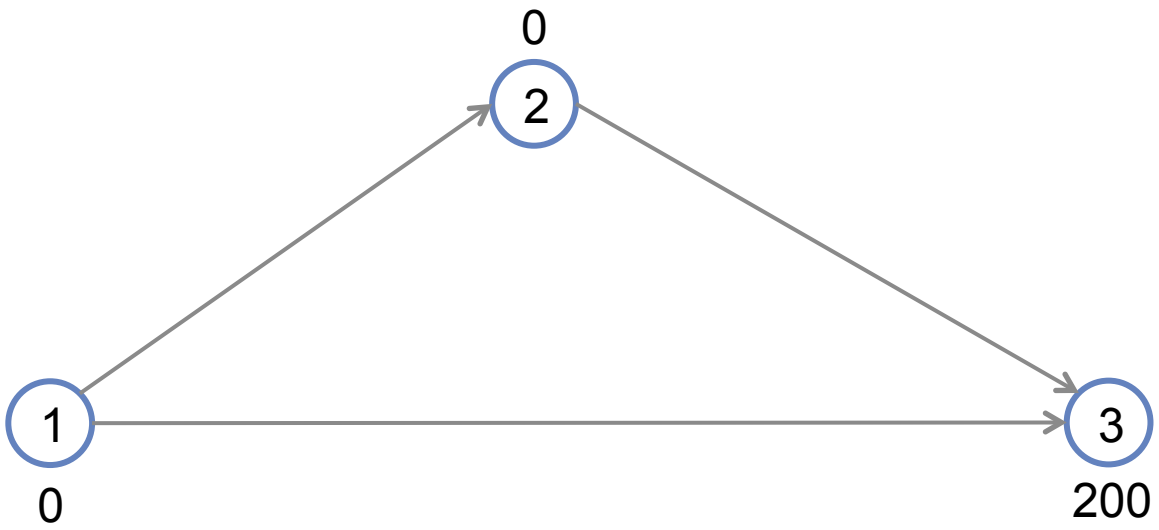
6．GraphX の Pregel 処理
④　影響の計測

【考え方】関係性の清算を行う



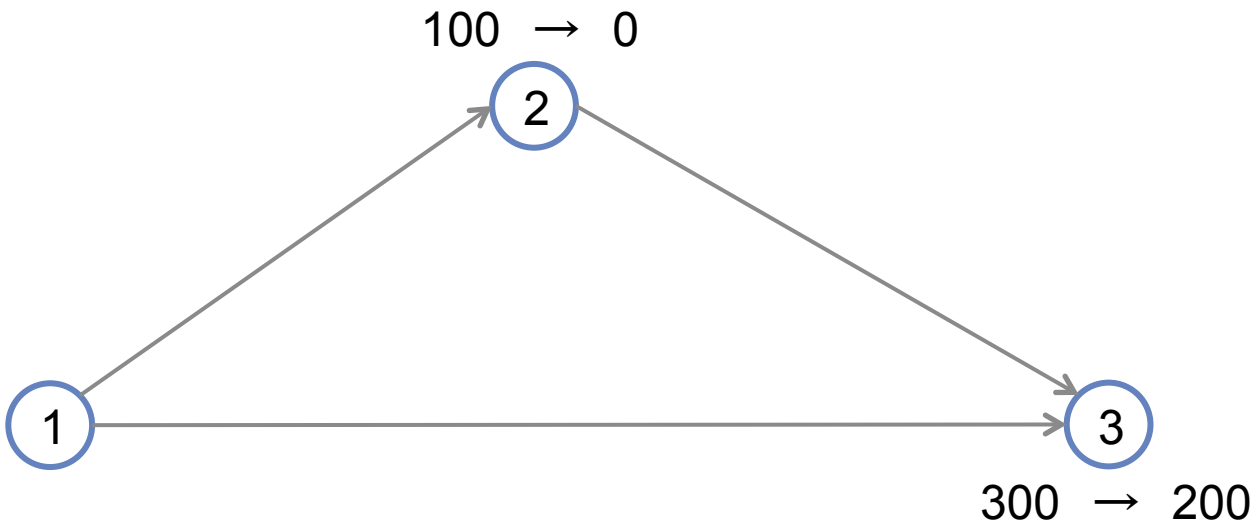
6．GraphX の Pregel 処理
④　影響の計測

【考え方】清算後の状態



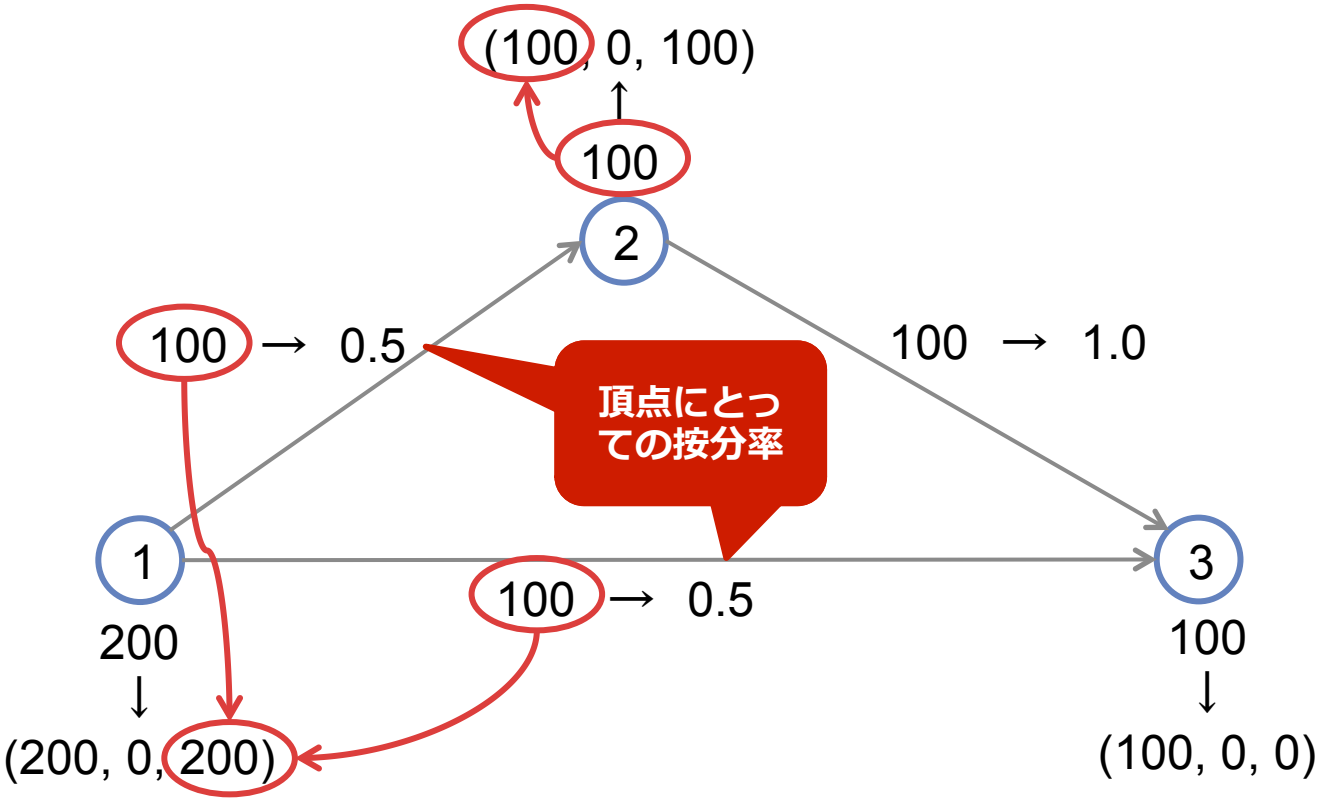
6．GraphX の Pregel 処理
④　影響の計測

【考え方】「1」の影響



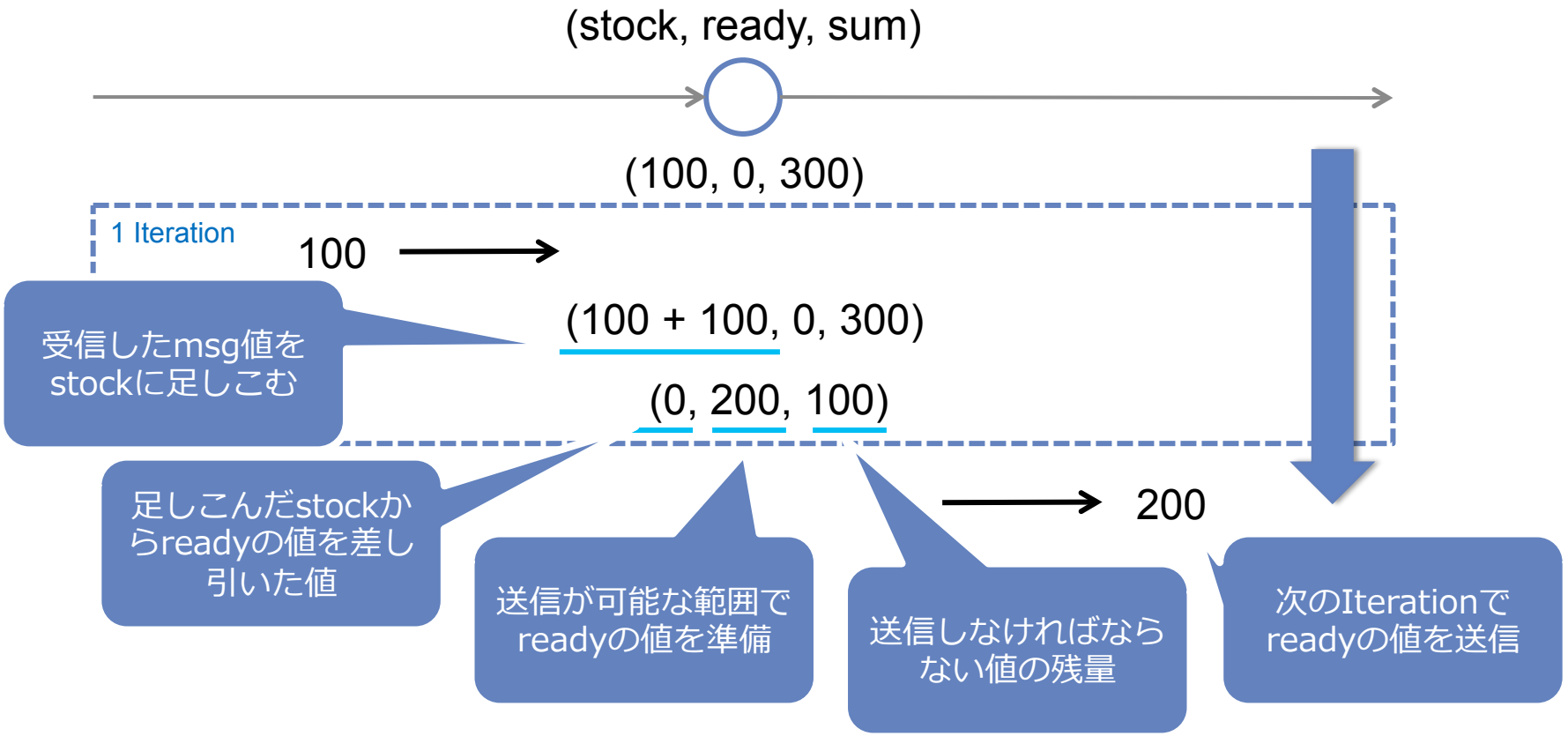
6．GraphX の Pregel 処理
④　影響の計測

【事前処理】下記のように頂点・辺の持つ値構造を変更する



6．GraphX の Pregel 処理
④　影響の計測

【実装イメージ】



6．GraphX の Pregel 処理
④　影響の計測

```
// 頂点毎に辺が持っている値を合計
val sumAttrList:VertexRDD[Long] = graph.mapReduceTriplets(
  mapFunc = edge => Iterator((edge.srcId, edge.attr)),
  reduceFunc = (a:Long, b:Long) => a + b)

// 頂点毎の辺の値の合計を、頂点に持たせる
val graph3:Graph[(Long, Long, Long, Long), Long]
 = graph.outerJoinVertices(sumAttrList){
  (vid:VertexId, oldAttr:Long, sumEdgesAttrOpt:Option[Long]) => {
   (oldAttr, 0, sumEdgesAttrOpt.getOrElse(0), oldAttr)}}

// 辺の値を、頂点毎の辺の値の合計に対する割合の値に変換する
val graph4:Graph[(Long, Long, Long, Long), Float]
 = graph3.mapTriplets(edge =>
    edge.attr.toFloat / edge.srcAttr._3.toFloat)
```

6．GraphX の Pregel 処理
④　影響の計測

```
val graph5:Graph[(Long, Long, Long, Long), Float] = Pregel(
 graph4, 0L, activeDirection = EdgeDirection.Out) (
 (id, attr, msg) => {
  val stock = attr._1, ready = attr._2,
              sum = attr._3, origin = attr._4
  val tmpStock = stock + msg
  val isEnough = if(tmpStock >= sum) true else false
  val afterSum = if(isEnough) 0 else sum - tmpStock
  val afterReady = if(isEnough) sum else tmpStock
  val afterStock = if(isEnough) tmpStock - sum else 0
  (afterStock, afterReady, afterSum, origin)},
 edge => {
  val stock = edge.srcAttr._1, ready = edge.srcAttr._2,
      sum = edge.srcAttr._3, origin = edge.srcAttr._4,
      rate = edge.attr
  if(ready <= 0) Iterator.empty
  else Iterator((edge.dstId, (ready * rate).toLong))},
 (a, b) => (a + b)
)
```

7．まとめ

1. 表構造で扱っていては不可能な分析が、グラフ構造データ分析であれば可能
2. ETL処理等との統一的な実装が、ひとつのシステム上で可能
3. Pregel によって行列計算をしないで、グラフデータの並列分散処理が可能

ご清聴ありがとうございました