SAMPLE CHAPTER

Spark GraphX IN ACTION

Michael S. Malak
Robin East

MANNING

Spark GraphX in Action

by Michael S. Malak

Robin East

Chapter 6

brief contents

6 Other useful graph algorithms

This chapter covers

- Standard graph algorithms that GraphX doesn't provide out of the box
- Shortest Paths on graphs with weighted edges
- The Traveling Salesman problem
- Minimum Spanning Trees

In chapter 5 you learned the foundational GraphX APIs that will enable you to write your own custom algorithms. But there's no need for you to reinvent the wheel in those cases where the GraphX API already provides an implemented standard algorithm. There are some algorithms that have been historically associated with graphs for decades but are not in the GraphX API. This chapter describes some of those classic graph algorithms and discusses which situations they can be used in.

These classic graph algorithms were invented in the 1950s, long before Spark or any other sort of parallel computing. They are iterative in nature—for example, they add one edge at a time to the solution. GraphX's Pregel API isn't a good match because it operates on all the vertices simultaneously. The power of GraphX's parallel processing is still being used, though, because each step in these algorithms

involves some kind of graph-wide search. You'll see how to use GraphX's iterative Map/Reduce facilities (`aggregateMessages()` together with `outerJoinVertices()`) to implement and parallelize these algorithms that were originally designed for serial computation.

The first of the three algorithms described in this chapter, Shortest Paths with Weights, fills a glaring hole in the GraphX API, which only provides a shortest-paths algorithm that assumes each edge has a weight of 1. Shortest Paths with Weights allows route planning on a map where each edge weight represents the distance between its two vertices (representing cities).

The second algorithm, called the Travelling Salesman, finds a path through a graph that hits every vertex. This algorithm is useful for package/mail delivery and other logistics applications.

The third and final algorithm, Minimum Spanning Tree, overlays a *tree* (a graph with no cycles) over the top of the graph where the sum of its edge weights is less than any other possible spanning tree. Although this sounds abstract (and is, in fact, one of the first algorithms presented in a graph theory course), it's useful for routing utilities and has other non-intuitive uses, such as creating hierarchical scientific or bibliographic taxonomies.

6.1 Your own GPS: Shortest Paths with Weights

Today, we take for granted the GPS capability in our smartphones and map apps. But how do they do it? Edsger Dijkstra figured it out in 1956, and this section implements a Spark version of that algorithm.

Section 5.3 showed GraphX's implementation of finding shortest-path lengths for graphs with unweighted edges, but Dijkstra's algorithm finds the shortest-path lengths for graphs with weighted edges (see figure 6.1). When way-finding on a geographical map, the vertices represent cities or road intersections, and the edge weights represent road distances.

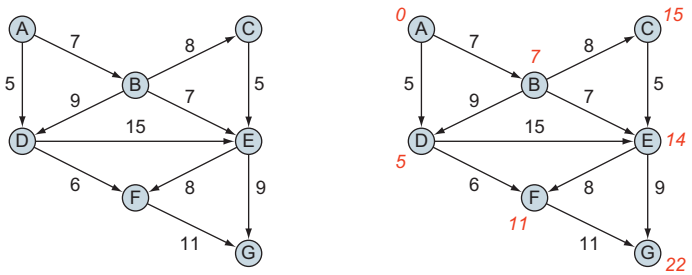

Figure 6.1 Example graph data and distances from vertex A after having been run through Dijkstra's algorithm. Given a graph with edge weights on the left, Dijkstra's algorithm annotates each vertex with a "shortest distance from vertex A." Graph data credit: the graph data comes from the Wikipedia article on Kruskal's algorithm (which, incidentally, is implemented in the last section of this chapter), which the contributor contributed to the public domain.

The Dijkstra algorithm calculates path distance from one particular vertex to every other vertex in the graph. It can be described like this:

1. Initialize the starting vertex to distance zero and all other vertices to distance infinity.
2. Set the current vertex to be the starting vertex.
3. For all the vertices adjacent to the current vertex, set the distance to be the lesser of either its current value or the sum of the current vertex's distance plus the length of the edge that connects the current vertex to that other vertex. For example, in figure 6.1, after the first iteration, vertex D has a value of 5, and vertex B has a value of 7. In the second iteration, there is a candidate alternative to get from A to D, which is through B, but that has a total path length of 16, so D keeps its old value of 5.
4. Mark the current vertex as having been visited.
5. Set the current vertex to be the unvisited vertex of the smallest distance value. If there are no more unvisited vertices, stop.
6. Go to step 3.

There are many variations of Dijkstra's algorithm, including versions for directed versus undirected graphs. The implementation in the following listing is geared toward directed graphs.

Listing 6.1 Dijkstra Shortest Paths distance algorithm

```
import org.apache.spark.graphx._
def dijkstra[VD](g:Graph[VD,Double], origin:VertexId) = {
  var g2 = g.mapVertices(
    (vid,vd) => (false, if (vid == origin) 0 else Double.MaxValue))

  for (i <- 1L to g.vertices.count-1) {
    val currentVertexId =
      g2.vertices.filter(!_._2._1)
        .fold((0L,(false,Double.MaxValue)))((a,b) =>
          if (a._2._2 < b._2._2) a else b)
        ._1

    val newDistances = g2.aggregateMessages[Double](
        ctx => if (ctx.srcId == currentVertexId)
                 ctx.sendToDst(ctx.srcAttr._2 + ctx.attr),
        (a,b) => math.min(a,b))

    g2 = g2.outerJoinVertices(newDistances)((vid, vd, newSum) =>
      (vd._1 || vid == currentVertexId,
       math.min(vd._2, newSum.getOrElse(Double.MaxValue))))
  }

  g.outerJoinVertices(g2.vertices)((vid, vd, dist) =>
    (vd, dist.getOrElse((false,Double.MaxValue))._2))
}
```

SPARK TIP The RDD API unfortunately doesn't include a `minBy()` function like regular Scala collections do, so a cumbersome and verbose `fold()` had to be used in the preceding code to accomplish the same thing.

In this implementation, we stoop to using a `var` (for `g2`) instead of a `val` because this is an iterative algorithm. When we initialize `g2`, we throw away any vertex data in the original `g` and attach our own: a pair of a Boolean and a Double. The Boolean indicates whether the vertex has been visited yet. The Double is the distance from the origin to that vertex.

As of GraphX 1.6 all graphs are immutable, so the only way to "update" these vertex values in our algorithm is to create a new graph. When we compute `newDistances`, we have to add that onto our graph `g2` with `outerJoinVertices()`, which creates a new graph. We assign that new graph back to `g2`, relying on JVM garbage collection to get rid of the old graph that was in `g2`.

As the last line of the function, which is the return value, we restore the original vertex properties by adding the final results from `g2` onto the original `g` with `outerJoinVertices`. In the process we make the type of the vertex properties for the return graph have an extra level of information; instead of `VD`, the vertex property type is a `Tuple2[VD,Double]`, where the Double contains the distance output from Dijkstra's algorithm.

The Pregel API would not have been easy to use due to the concept of the "current vertex," which for each iteration is the global overall minimum. The Pregel API is more suited for algorithms that treat all vertices as equals. The next listing shows how to execute our new `dijkstra()` function with the graph from figure 6.1.

Listing 6.2 Executing the Shortest Path distance algorithm

```
val myVertices = sc.makeRDD(Array((1L, "A"), (2L, "B"), (3L, "C"),
  (4L, "D"), (5L, "E"), (6L, "F"), (7L, "G")))
val myEdges = sc.makeRDD(Array(Edge(1L, 2L, 7.0), Edge(1L, 4L, 5.0),
  Edge(2L, 3L, 8.0), Edge(2L, 4L, 9.0), Edge(2L, 5L, 7.0),
  Edge(3L, 5L, 5.0), Edge(4L, 5L, 15.0), Edge(4L, 6L, 6.0),
  Edge(5L, 6L, 8.0), Edge(5L, 7L, 9.0), Edge(6L, 7L, 11.0)))
val myGraph = Graph(myVertices, myEdges)

dijkstra(myGraph, 1L).vertices.map(_._2).collect

res0: Array[(String, Double)] = Array((D,5.0), (A,0.0), (F,11.0), (C,15.0),
 (G,22.0), (E,14.0), (B,7.0))
```

These are the values shown in figure 6.1. But wait a minute—how would you know the path sequence to get to any of these destination vertices? The algorithm computes the distances, but not the paths.

The following listing adds the common embellishment of keeping track of the paths every step of the way by adding a third component to the vertex tuple, a Scala List that accumulates breadcrumbs.

Listing 6.3 Dijkstra's Shortest Path algorithm with breadcrumbs

```
import org.apache.spark.graphx._
def dijkstra[VD](g:Graph[VD,Double], origin:VertexId) = {
  var g2 = g.mapVertices(
    (vid,vd) => (false, if (vid == origin) 0 else Double.MaxValue,
                 List[VertexId]()))

  for (i <- 1L to g.vertices.count-1) {
    val currentVertexId =
      g2.vertices.filter(!_._2._1)
        .fold((0L,(false,Double.MaxValue,List[VertexId]())))((a,b) =>
           if (a._2._2 < b._2._2) a else b)
        ._1

    val newDistances = g2.aggregateMessages[(Double,List[VertexId])](
        ctx => if (ctx.srcId == currentVertexId)
                 ctx.sendToDst((ctx.srcAttr._2 + ctx.attr,
                                ctx.srcAttr._3 :+ ctx.srcId)),
        (a,b) => if (a._1 < b._1) a else b)

    g2 = g2.outerJoinVertices(newDistances)((vid, vd, newSum) => {
      val newSumVal =
        newSum.getOrElse((Double.MaxValue,List[VertexId]()))
      (vd._1 || vid == currentVertexId,
       math.min(vd._2, newSumVal._1),
       if (vd._2 < newSumVal._1) vd._3 else newSumVal._2)})
  }

  g.outerJoinVertices(g2.vertices)((vid, vd, dist) =>
    (vd, dist.getOrElse((false,Double.MaxValue,List[VertexId]()))
             .productIterator.toList.tail))
}
```

SCALA TIP The "operator" (Scala purists call them *functions*, even though they look like operators) `:+` is a Scala List function that returns a new list with an element appended. Scala List has a large number of similar operators for prepending or appending lists or elements, and these are listed along with the other Scala List functions in the Scaladocs for List.

Listing 6.4 Executing the Shortest Path algorithm that uses breadcrumbs

```
dijkstra(myGraph, 1L).vertices.map(_._2).collect
```

```
res1: Array[(String, List[Any])] = Array((D,List(5.0, List(1))),
 (A,List(0.0, List())), (F,List(11.0, List(1, 4))),
 (C,List(15.0, List(1, 2))), (G,List(22.0, List(1, 4, 6))),
 (E,List(14.0, List(1, 2))), (B,List(7.0, List(1))))
```

That's much better. Now you can know the shortest path to take to get to any of the other vertices.

6.2 Travelling Salesman: greedy algorithm

The travelling salesman problem tries to find the shortest path through an undirected graph that hits every vertex. For example, if a salesperson needs to visit every city in a region, they would like to minimize the total distance traveled.

Unlike the shortest path problem in the previous section, there is no easy, straightforward, deterministic algorithm to solve the travelling salesman problem. Note that *travelling salesman* is a well-known math problem; the term was coined in the 1930s, so that term is used here rather than inclusive language.

The problem is of a class of problems known as *NP-hard*, which means it can't be solved in an amount of time that is a polynomial with respect to the number of vertices or edges. It is, rather, a combinatorial optimization problem that would require an exponential amount of time to solve optimally. Instead of trying to find the optimum, various approaches use heuristics to come close to the optimum. The implementation shown in figure 6.2 uses the greedy algorithm, which is the simplest algorithm but it also gives answers that can be far from optimal and don't necessarily hit all the vertices. (If hitting every vertex is a requirement, the algorithm might produce no acceptable answer at all.) This algorithm is called *greedy* because at every iteration it grabs the immediate shortest edge without doing any kind of deeper search.

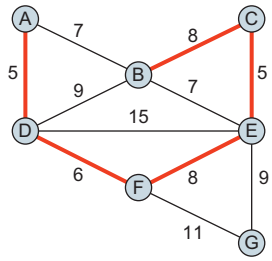

Figure 6.2 The greedy approach to the Travelling Salesman problem is the simplest, but it doesn't always hit all the vertices. In this example, it neglected to hit vertex G.

The greedy algorithm can be improved without much additional coding by iterating and rerunning the whole algorithm with different starting vertices, picking from the resulting solutions the one that goes to all the vertices and is the shortest. But the implementation shown in listing 6.5 only does one execution of the greedy algorithm for a given starting vertex passed in as a parameter.

The approach of the greedy algorithm is simple:

1. Start from some vertex.
2. Add the adjacent edge of lowest weight to the spanning tree.
3. Go to step 2.

Listing 6.5 Travelling Salesman greedy algorithm

```
def greedy[VD](g:Graph[VD,Double], origin:VertexId) = {
  var g2 = g.mapVertices((vid,vd) => vid == origin)
            .mapTriplets(et => (et.attr,false))
  var nextVertexId = origin
  var edgesAreAvailable = true

  do {
    type tripletType = EdgeTriplet[Boolean,Tuple2[Double,Boolean]]

    val availableEdges =
```

```
      g2.triplets
        .filter(et => !et.attr._2
                      && (et.srcId == nextVertexId && !et.dstAttr
                          || et.dstId == nextVertexId && !et.srcAttr))

    edgesAreAvailable = availableEdges.count > 0

    if (edgesAreAvailable) {
      val smallestEdge = availableEdges
          .min()(new Ordering[tripletType]() {
             override def compare(a:tripletType, b:tripletType) = {
               Ordering[Double].compare(a.attr._1,b.attr._1)
             }
           })

      nextVertexId = Seq(smallestEdge.srcId, smallestEdge.dstId)
                     .filter(_ != nextVertexId)(0)

      g2 = g2.mapVertices((vid,vd) => vd || vid == nextVertexId)
             .mapTriplets(et => (et.attr._1,
                                 et.attr._2 ||
                                   (et.srcId == smallestEdge.srcId
                                    && et.dstId == smallestEdge.dstId)))
    }
  } while(edgesAreAvailable)

  g2
}

greedy(myGraph,1L).triplets.filter(_.attr._2).map(et=>(et.srcId, et.dstId))
                  .collect

res1: Array[(org.apache.spark.graphx.VertexId,
 org.apache.spark.graphx.VertexId)] = Array((1,4), (2,3), (3,5), (4,6),
 (5,6))
```

> **SCALA TIP** `type` is a convenient way to alias types at compile time to prevent having to type out long types over and over again. It's similar to `typedef` in C/C++, but there's no equivalent in Java. `type` in Scala also has another use—to introduce abstract type members in `traits`—but that's beyond the scope of this book.

Here we stoop to using three `vars`. The third one is for loop control because Scala has no `break` keyword (although there is a simulation of `break` in the standard library). In this implementation, during the looping the graph `g2` has different vertex property types and a different edge property type than for `g` that was passed in. The vertex property type is Boolean, indicating whether the vertex has been incorporated into the solution yet. The edge property type is similar, except the edge weight is carried along as well because it's used by the algorithm; specifically, the `Edge` attribute type is `Tuple2[Double,Boolean]` where the Double is the edge weight and the Boolean indicates whether the edge has been incorporated as part of the solution.

The computation of `availableEdges` checks edges in both directions. This is what makes the algorithm treat graphs as if they were undirected. All GraphX graphs are directed in reality, with a source and a destination. Any implementation in GraphX of

other algorithms meant for undirected graphs would have to take similar precautions of checking both directions.

Similar to the Shortest Paths algorithm in the previous section, we had to create a new graph at the end of the iteration due to the immutability of graphs in GraphX. For each iteration, the algorithm needs to set the Boolean on one vertex and the Boolean on one edge, but nevertheless, a new graph has to be created. Also similar to the Shortest Paths algorithm, the Pregel API would not be a good choice here because the greedy Travelling Salesman algorithm adds one edge at a time (to one vertex); it isn't treating all vertices equally.

The resulting graph finally returned by greedy is in perhaps not the most convenient form for the caller. We didn't bother to glom back on the original vertex properties from g. The reason we didn't is that we would have had to join the edge properties from `g` and `g2` together, and GraphX provides no automatic way to do that. We do it in the next section, but it requires more than a line of code, which is a lot for Scala.

6.3 Route utilities: Minimum Spanning Trees

Minimum Spanning Trees sound abstract and not useful, but they can be considered to be like the Travelling Salesman problem where you don't care if you have to backtrack (and where backtracking is free of cost). One of the most immediate applications is routing utilities (roads, electricity, water, and so on) to ensure that all cities receive the utility, at minimum cost (for example, minimum distance, if the edge weights represent distance between cities). There are some non-obvious applications of Minimum Spanning Trees as well, including the creation of taxonomies among a collection of similar items, such as animals (for scientific classification) or newspaper headlines. See figure 6.3.

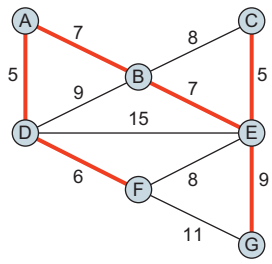

Figure 6.3 A Minimum Spanning Tree is a tree (a graph with no cycles) that covers every vertex of an undirected graph, of minimum total weight (sum of the edge weights).

Listing 6.6 is an implementation of Kruskal's algorithm. Again, the example graph used throughout this chapter is the same as the example graph on the Wikipedia page of Kruskal's algorithm, and because that Wikipedia page illustrates the execution of the algorithm through a sequence of graph illustrations, you can see exactly how the following implementation works.

Even though Kruskal's algorithm is greedy, it does find one of the Minimum Spanning Trees (there may be more than one Spanning Tree that has the same total weight). Finding a Minimum Spanning Tree isn't a combinatorial problem. Kruskal's algorithm is called greedy because at every iteration it grabs the edge of lowest weight. Unlike the Travelling Salesman greedy algorithm, the result is mathematically provably a Minimum Spanning Tree. See figure 6.4.

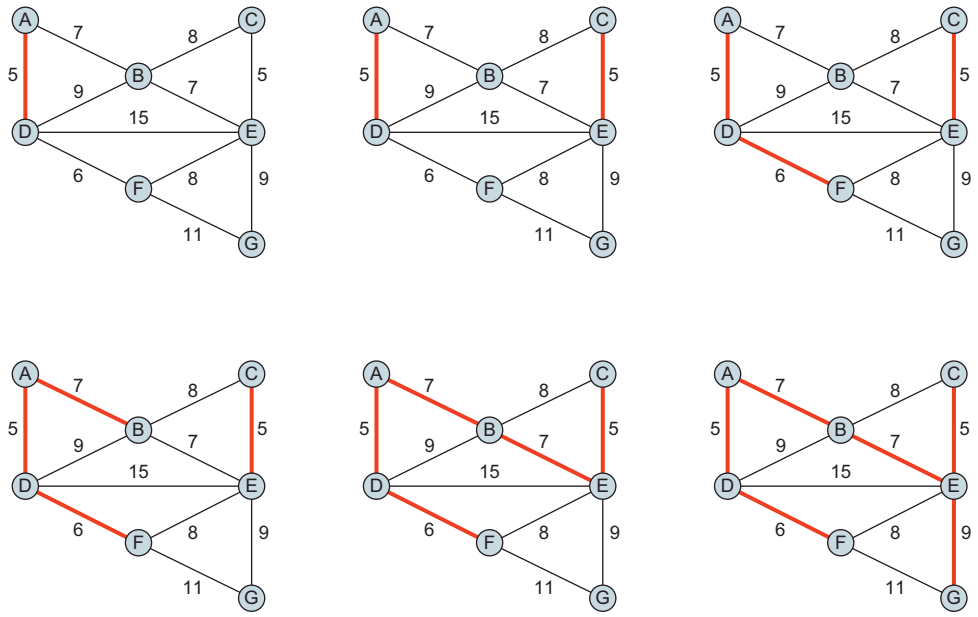

Figure 6.4 Iteration steps of Kruskal's algorithm to find a Minimum Spanning Tree. In each iteration, the whole graph is searched for the unused edge of lowest weight. But there's a catch: that edge can't form a cycle (as a *tree* is what is being sought).

Unlike the Travelling Salesman algorithm in the previous section, Kruskal's algorithm doesn't build up the tree by extending out an edge at a time from some growing tree. Rather, it does a global search throughout the graph to find the edge with the least weight to add to the set of edges that will eventually form a tree. The algorithm can be described like this:

1. Initialize the set of edges that will eventually comprise the resulting minimum spanning tree to be empty.
2. Find the edge of smallest weight throughout the whole graph that meets the following two conditions and add it to the result set:
   a. The edge isn't already in the result set of edges.
   b. The edge doesn't form a cycle with the edges already in the result set of edges.
3. Go to step 2, unless all vertices are already represented in the result set of edges.

The second condition (b) in step 2 is the tricky one. Finding a cycle is easy for a human to comprehend, but it's not immediately obvious how to describe it to a computer. There are a few approaches we could have taken. We could find the shortest path (for example, by invoking GraphX's built-in `ShortestPaths`, described in section 5.3) for every candidate edge (which is all or almost all of them in the beginning) and then discard from consideration those edges whose vertices already have a path between them. Another approach, which is the one taken in listing 6.6, is to call GraphX's built-in `connectedComponents()`, described in section 5.4. This gives vertex

connectivity information across the whole graph in one fell swoop. If the two vertices of an edge belong to the same connected component of the result tree so far, then that edge is ineligible for consideration because adding it to the result set would create a cycle.

Listing 6.6 Minimum Spanning Tree

```
def minSpanningTree[VD:scala.reflect.ClassTag](g:Graph[VD,Double]) = {
  var g2 = g.mapEdges(e => (e.attr,false))

  for (i <- 1L to g.vertices.count-1) {
    val unavailableEdges =
      g2.outerJoinVertices(g2.subgraph(_.attr._2)
                             .connectedComponents
                             .vertices)((vid,vd,cid) => (vd,cid))
        .subgraph(et => et.srcAttr._2.getOrElse(-1) ==
                        et.dstAttr._2.getOrElse(-2))
        .edges
        .map(e => ((e.srcId,e.dstId),e.attr))

    type edgeType = Tuple2[Tuple2[VertexId,VertexId],Double]

    val smallestEdge =
      g2.edges
        .map(e => ((e.srcId,e.dstId),e.attr))
        .leftOuterJoin(unavailableEdges)
        .filter(x => !x._2._1._2 && x._2._2.isEmpty)
        .map(x => (x._1, x._2._1._1))
        .min()(new Ordering[edgeType]() {
           override def compare(a:edgeType, b:edgeType) = {
             val r = Ordering[Double].compare(a._2,b._2)
             if (r == 0)
               Ordering[Long].compare(a._1._1, b._1._1)
             else
               r
           }
         })

    g2 = g2.mapTriplets(et =>
      (et.attr._1, et.attr._2 || (et.srcId == smallestEdge._1._1
                                  && et.dstId == smallestEdge._1._2)))
  }

  g2.subgraph(_.attr._2).mapEdges(_.attr._1)
}

minSpanningTree(myGraph).triplets.map(et =>
 (et.srcAttr,et.dstAttr)).collect

Res2: Array[(String, String)] = Array((A,B), (A,D), (B,E), (C,E), (D,F),
 (E,G))
```

SCALA TIP Sometimes when using Scala generics it's necessary to declare the type parameters to be of `scala.reflect.ClassTag`. This is due to JVM type-erasure at runtime. In listing 6.6, the type VD is needed at runtime and not only at compile-time for the call to `subgraph()`.

In this implementation, the only type change we make to the graph during iteration is to add a Boolean to the `Edge` properties to indicate whether that edge is part of the result set of edges for the Spanning Tree.

All the tricky magic happens in the assignment to `unavailableEdges`. After first subsetting the graph to be those edges already in the growing result set of edges, we run it through `connectedComponents()`. We then take those component IDs and glom them onto the regular vertex data with `outerJoinVertices()`. Then we say that an edge is unavailable if its two vertices belong to the same connected component. Those component IDs could be conceptually `null` (in Scala-speak, `Option[VertexId]` could be `None`) due to the way `outerJoinVertices()` works. If a vertex isn't part of the growing result set of edges, then it's free and clear, and we prevent such edges from being declared unavailable by saying `None` is equivalent to a component ID of –1 or –2. –1 and –2 were intentionally chosen to be not only invalid `vertexId`s but also different from each other so that an edge with both vertices not already part of the growing result set of edges would be considered still available.

The computation of `smallestEdge` contains that conceptual join on edges that we said earlier in this chapter was messy because GraphX doesn't have an edge join built in. We'll convert the edges to a `Tuple2[Tuple2[VertexId, VertexId], Tuple2[Double, Boolean]]` and then use the regular RDD `leftOuterJoin()` (not to be confused with the GraphX-specific `join()`s and `outerJoin()`s). `leftOuterJoin()` will treat the `Tuple2[VertexId, VertexId]` as a single entity and key off that when it performs the join. After the `leftOuterJoin()`, the data type has five parts and looks like figure 6.5.

After removing the edges unavailable for consideration, the subsequent `map()` then extracts only the two pieces of information we care about: the pair of `VertexId`s and the edge weight. The `override def compare` first compares the edge weights, and if those are equal, breaks the tie by comparing to see which `VertexId` is less. This is to make the execution deterministic and repeatable and to match the results in the

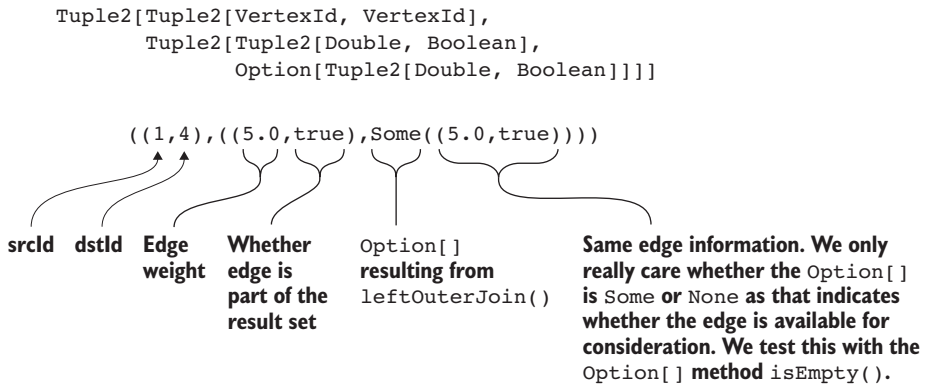

Figure 6.5 The data type after the `leftOuterJoin()` of listing 6.6.

Wikipedia example, because that is apparently how they broke ties. If you don't care about repeatability or matching Wikipedia (if you don't care about trying to make sure your program is matching that known working implementation), you can replace the `override def compare` body with `Ordering[Double].compare(a._2,b._2)`.

The return value of `minSpanningTree()` is the tree itself rather than the whole graph. The edge property type (`Edge` attribute) is restored to be the weight; the temporary Boolean is stripped off.

6.3.1 Deriving taxonomies with Word2Vec and Minimum Spanning Trees

One way to look at Minimum Spanning Trees is to see them as extracting (in some sense) the most important connections in the graph. By removing the less important edges we make the graph sparser, reducing it to its essentials. This section shows you how to use machine learning and graph processing to turn a simple list of unconnected terms—in this case, a list of animal names—into a connected taxonomy using Minimum Spanning Trees (MSTs).

MSTs can't do all the work, though; we'll also get some help from a natural language-processing tool called Word2Vec. Word2Vec lets us assign distances between each of our terms so that we can build a weighted graph of the connections between the terms. We can then run Minimum Spanning Tree on the graph to reveal the most important connections.

Understanding Word2Vec

Word2Vec is a natural language-processing algorithm that turns a text corpus (a collection of text documents) into a set of *n*-dimensional vectors that represent each distinct word in the corpus. Each word in the corpus is represented by a vector in the set. See figure 6.6.

What's useful about the vectors that Word2Vec generates is that words that are semantically similar tend to be close together. We can use a measure of similarity called *cosine similarity* to assign a number to how similar those words are. Cosine similarity

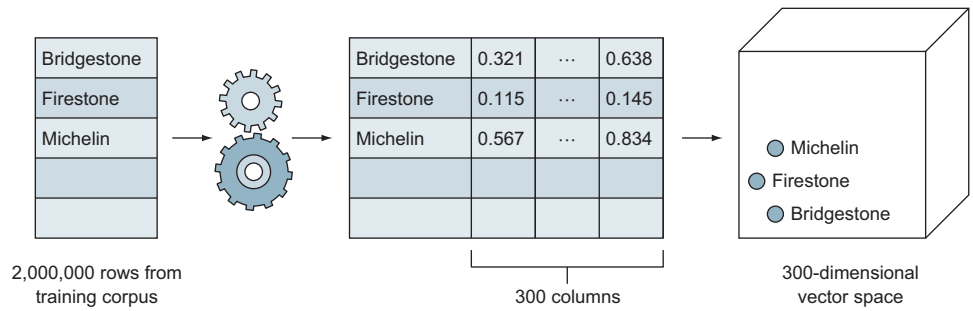

Figure 6.6 Words are extracted from the training corpus and processed by the Word2Vec algorithm to produce an *n*-dimensional vector for each word—here, *n* is 300. Semantically similar words, such as *Michelin* and *Firestone*, are close together.

ranges from 1 (similar) through 0 (not similar) to –1. We then turn cosine similarity into cosine distance by subtracting from 1:

```
cosine distance = 1 - cosine similarity
```

Now we have a measure where similar words have small distances between them and less similar words have larger distances.

Models trained with Word2Vec usually require large amounts of data to be effective. Luckily there are already a number of well-regarded pre-trained models that can be used. For this task we'll use the model trained on a subset of Google News (100 billion words). Even though Spark's machine learning library, MLlib, contains an implementation of the Word2Vec algorithm, it doesn't yet have the functionality to load the binary format Google News model. Instead, we've pre-calculated cosine distance for every pair of names in the animal list and stored the results in a comma-separated variable file. The file, called animal_distances.txt, is included as part of the code download for this chapter. If you open up the file, you should see the following in the first few lines:

```
sea_otter,sea_otter,-0.000000
sea_otter,animal,0.638965
sea_otter,chicken,0.860217
sea_otter,dog,0.705229
sea_otter,aardvark,0.767667
sea_otter,albatross,0.770162
```

The file is in comma-separated format with three columns of data. Each row contains a single pair of terms and their cosine distance. There are around 224 different animal names in the list.

Creating the distances file

Because Spark MLlib doesn't yet have the ability to load Word2Vec models created by other implementations of Word2Vec, we use the Python library Gensim. If you want to use Word2Vec to generate the distance file yourself, you will need to follow the installation instructions at https://radimrehurek.com/gensim/install.html. Typically this involves using easy_install

```
easy_install -U gensim
```

or pip:

```
pip install --upgrade gensim
```

Then you will need to download the Google News model from the link on the page at https://code.google.com/p/word2vec/. Be warned: the file is several GBs in size.

Now we are ready to generate the distances. We start with a list of 200 animal terms that are in the file animal_terms.txt in the code download for this chapter:

```
from gensim.models import Word2Vec
model = Word2Vec.load_word2vec_format(
```

```
    'GoogleNews-vectors-negative300.bin',
    binary=True)
f = open('animal_terms.txt')
animals = f.read().splitlines()
animals = [x.lower() for x in animals if x.lower() in
            model.vocab.keys()]
f.close()
f = open('animal_distances.txt','w')
f.truncate()
for i in range(0, len(animals)):
  for j in range(i, len(animals)):
    f.write('%s,%s,%1.6f\n' %
        (animals[i], animals[j],
        1 - model.similarity(animals[i], animals[j])))
f.flush()
f.close()
```

The output is the file animal_distances.txt that is used to build the distances graph.

Creating the Minimum Spanning Tree

We now use the list to build a graph of the connections between each animal based on the distances derived from the GoogleNews Word2Vec model.

DEFINITION A complete graph is one where every vertex has an edge with every other vertex. The number of edges e in the graph as a function of the number of vertices v is $e = n(n-1)/2$

Each animal is a vertex, and the connections between animals are weighted edges. Each edge corresponds to the cosine distance between the vector representation of each animal. Because we generate an edge for every pair of animals, our graph is complete.

Listing 6.7 shows the code to build the graph and generate the Minimum Spanning Tree. We use the `toGexf()` method developed in chapter 4 to write the tree to a file that can be opened for visualization by Gephi. With more than 200 vertices, the tree is rather big, so we show a portion of the graph in figure 6.7.

Listing 6.7 Building the distances graph

```
val dist = sc.textFile("animal_distances.txt")
val verts = dist.map(_.split(",")(0)).distinct.
    map(x => (x.hashCode.toLong,x))
val edges = dist.map(x => x.split(",")).
    map(x => Edge(x(0).hashCode.toLong,
                  x(1).hashCode.toLong,
                  x(2).toDouble))
```

animal_distances.txt has animal names in columns 0 and 1, we choose column 0 and apply distinct method to ensure unique vertices.

Calling hashCode and converting to Long is one way to allocate a VertexID to each vertex.

```
val distg = Graph(verts, edges)
val mst = minSpanningTree(distg)
val pw = new java.io.PrintWriter("animal_taxonomy.gexf")
pw.write(toGexf(mst))
pw.close
```

Call our Minimum Spanning Tree algorithm!

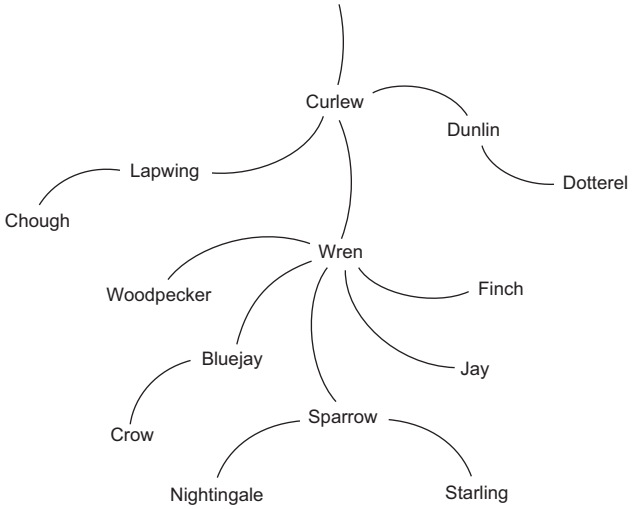

Figure 6.7 A section of the animal taxonomy Minimum Spanning Tree showing connections between birds.

This example shows how we can derive accurate semantic connections between a set of terms. We used an example of animal names, but a similar approach could be used for other areas with large corpuses of unstructured text, such as medical literature or reports of companies listed on global stock exchanges.

6.4 Summary

- Many of the classic graph algorithms don't lend themselves to implementation with Pregel. We looked at custom implementations of Shortest Paths with Weights, Travelling Salesman, and Minimum Spanning Tree.
- A Minimum Spanning Tree "sparsifies" a graph, reducing it to its essentials.
- Spark allows you to easily combine graph processing with other machine learning algorithms, as we showed in the creation of an animal taxonomy.

DATA SCIENCE/SCALA

Spark GraphX IN ACTION

Malak • East

GraphX is a powerful graph processing API for the Apache Spark analytics engine that lets you draw insights from large datasets. GraphX gives you unprecedented speed and capacity for running massively parallel and machine learning algorithms.

Spark GraphX in Action begins with the big picture of what graphs can be used for. This example-based tutorial teaches you how to use GraphX interactively. You'll start with a crystal-clear introduction to building big data graphs from regular data, and then explore the problems and possibilities of implementing graph algorithms and architecting graph processing pipelines. Along the way, you'll collect practical techniques for enhancing applications and applying machine learning algorithms to graph data.

What's Inside

- Understanding graph technology
- Using the GraphX API
- Developing algorithms for big graphs
- Machine learning with graphs
- Graph visualization

Readers should be comfortable writing code. Experience with Apache Spark and Scala is not required.

Michael Malak has worked on Spark applications for Fortune 500 companies since early 2013. **Robin East** has worked as a consultant to large organizations for over 15 years and is a data scientist at Worldpay.

"Learn complex graph processing from two experienced authors… A comprehensive guide."
—Gaurav Bhardwaj, 3Pillar Global

"The best resource to go from GraphX novice to expert in the least amount of time."
—Justin Fister, PaperRater

"A must-read for anyone serious about large-scale graph data mining!"
—Antonio Magnaghi
OpenMail

"Reveals the awesome and elegant capabilities of working with linked data for large-scale datasets."
—Sumit Pal
Independent consultant

To download their free eBook in PDF, ePub, and Kindle formats, owners of this book should visit
www.manning.com/books/spark-graphx-in-action

ISBN 13: 978-1-61729-252-1
ISBN 10: 1-61729-252-4
54999
9 781617 292521

MANNING $49.99 / Can $57.99 [INCLUDING eBOOK]